CrossTalk

June 2007 **The Journal of Defense Software Engineering** Vol. 20 No. 6

# COTS Integration

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*



| | | |
|---|---|---|
| 1. REPORT DATE<br>**JUN 2007** | 2. REPORT TYPE | 3. DATES COVERED<br>**00-00-2007 to 00-00-2007** |
| 4. TITLE AND SUBTITLE<br>**CrossTalk: The Journal of Defense Software Engineering. Volume 20, Number 6, June 2007** | | 5a. CONTRACT NUMBER<br>5b. GRANT NUMBER<br>5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER<br>5e. TASK NUMBER<br>5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>**OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR'S ACRONYM(S)<br>11. SPONSOR/MONITOR'S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT<br>**Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT<br>**unclassified** | b. ABSTRACT<br>**unclassified** | c. THIS PAGE<br>**unclassified** | **Same as Report (SAR)** | **32** | |

# COTS Integration



# Software Engineering Technology




**On The Cover**

Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

# Departments




# CrossTalk

| | |
|---|---|
| **Co-Sponsors:** | |
| **DoD-CIO** | *The Honorable John Grimes* |
| **NAVAIR** | *Jeff Schwalb* |
| **76 SMXG** | *Kevin Stamey* |
| **309 SMXG** | *Randy Hill* |
| **402 SMXG** | *Diane Suchan* |
| **DHS** | *Joe Jarzombek* |
| **Staff:** | |
| **Managing Director** | *Brent Baxter* |
| **Publisher** | *Elizabeth Starrett* |
| **Managing Editor** | *Kase Johnstun* |
| **Associate Editor** | *Chelene Fortier-Lozancich* |
| **Article Coordinator** | *Nicole Kentta* |
| **Phone** | (801) 775-5555 |
| **E-mail** | crosstalk.staff@hill.af.mil |
| **CrossTalk Online** | www.stsc.hill.af.mil/crosstalk |

**CrossTalk, The Journal of Defense Software Engineering** is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); U.S. Navy (USN); U.S. Air Force (USAF); Defense Finance and Accounting Services (DFAS); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); Ogden-ALC 309 SMXG; and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

**The USAF Software Technology Support Center (STSC)** is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

***Subscriptions***: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 15.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

***Article Submissions***: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.





***CrossTalk Online Services:*** See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

***Back Issues Available:*** Please phone or e-mail us to see if back issues are available free of charge.

# Navigating the COTS Sea

As I scratch my head for the thousandth time wondering who came up with the bright idea of the standard desktop computer, it occurs to me that this month's CrossTalk theme is extremely pertinent. I am sure that the promoters of the standard desktop did not consider that a software maintenance group might need multiple versions of a single software package loaded onto the same desktop. Nor did they imagine that continually pushing out patches would create a configuration management nightmare in our software integration laboratories.

Commercial off-the-shelf (COTS) software solutions have long been touted as the best- or least-cost solution to many software design requirements. We save major development costs by using commercial products and, in theory, can significantly accelerate the delivery schedule. The Air Force is engaging in this venture on a massive scale with the implementation of Expeditionary Combat Support System (ECSS) – based on a commercially available Enterprise Resource Planning system. The projected life-cycle cost savings due to phasing out legacy information systems are staggering. However, the success of ECSS, and of all COTS software implementations, is dependent on a realistic assessment of all costs, benefits, and risks.

While I have never led a successful COTS implementation, I've witnessed quite a few failures and participated in many re-vectoring efforts. The following is some advice for anyone starting down that road:

**1. Remember the Titanic.** Don't just look at the tip of the iceberg, there's plenty hidden beneath the water to help sink the ship. Look at long-term license and maintenance fees. Check interfaces to see how many will have to be changed if you field a COTS solution. Expect interface control documents to have discrepancies. Ensure that you have sufficient data rights to maintain the product once it is fielded.

**2. Caveat emptor: Buyer beware!** Do your research, define your requirements, establish acceptance criteria, read the fine print and remember that the vendor is selling a product and is rewarded based on the number of sales. Ask for references from other customers who have successfully converted to the product.

**3. Resist the temptation to modify COTS.** If you start talking about GOTS (government off-the-shelf) and MOTS (modified off-the-shelf), then you've lost the bubble and should seek professional help. This isn't Burger King. If you go with COTS, you can't have it your way.

This month's articles illustrate both the drawbacks and the benefits of using COTS software in the development and sustainment of Department of Defense weapon systems. In *Added Sources of Costs in Maintaining COTS-Intensive Systems*, Dr. Betsy Clark and Dr. Brad Clark capture the many frustrations expressed by project managers and team leads in communicating to upper management the reasons why COTS-based systems are so expensive to maintain. Dr David A. Cook addresses *Issues to Consider Before Acquiring COTS*, in particular the problems of trying to integrate multiple COTS applications. Essential to success is the need to apply basic software engineering principles and beware of marketing hype.

At the 402 Software Maintenance Group at Robins AFB, we continue to see benefits from using COTS software, particularly in our integration environments if used wisely. In *Lean AISF: Applying COTS to System Integration Facilities*, Harold Lowery emphasizes the importance of weighing all alternatives and clearly defining trade-offs when considering make-versus-buy decisions. Additionally, in *GL Studio Brings Realism to Aircraft Cockpit Simulator Displays*, Kim Stults demonstrates that with sufficient pre-planning, COTS products can be a viable means of upgrading systems saving both time and budget. We conclude the theme articles with a discussion of using JAVA with real-time systems in *Applying COTS Java Benefits to Mission-Critical Real-Time Software* by Dr. Kelvin Nilsen.

The debate on the benefits and challenges of COTS is not likely to be settled soon. Most things in life have their benefits and drawbacks; as with life, we must keep an open mind when considering COTS.

Diane E Suchan

Diane E. Suchan
*Warner Robins Air Logistics Center Co-Sponsor*

# Added Sources of Costs in Maintaining COTS-Intensive Systems

Dr. Betsy Clark and Dr. Brad Clark
*Software Metrics Inc.*

*Ten years ago, work was begun at the Center for Systems and Software Engineering at the University of Southern California to develop a cost model for commercial off-the-shelf (COTS)-based software systems[1]. A series of interviews were conducted to collect data to calibrate this model[2]. A total of 25 project managers were interviewed; for eight of these projects, data was collected during the original system development and maintenance phases. A common sentiment heard from the people maintaining these systems was that they turned out to be more expensive to maintain than originally envisioned and, in fact, were more costly than a comparable custom-built system. At the same time, several people expressed frustration about the difficulty of communicating to upper management the reasons why COTS-based systems were so expensive to maintain. Anecdotal evidence from these interviews is used to discuss the added sources of maintenance cost. Three different approaches or strategies for system maintenance were observed and are summarized in this article.*

The past 10 to 15 years have seen a strong push within the Department of Defense and other government agencies toward the use of COTS software products in system acquisition. On the surface, this makes a lot of sense – why build something from scratch that already exists, especially if it is a mature product? In fact, with the increasing complexity of today's systems, a total custom system is no longer practical. With the continued use of COTS components in building systems, it is a worthwhile objective to identify sources of cost and approaches to managing them. It is the intent of this article to identify these sources.

As part of an effort to collect data to calibrate a cost model for systems consisting of COTS components, a number of interviews were conducted with project managers, team leads, and other project members maintaining COTS-intensive systems. People consistently told us that these systems were more expensive to maintain than originally estimated and, in fact, were more costly than a comparable custom-built system to maintain. At the same time, we heard frustration expressed about the difficulty of communicating to upper management the reasons why COTS-based systems are so expensive to maintain. If COTS-based systems really are more costly to maintain, what are these additional costs? Are there strategies for managing or minimizing them? These questions are addressed in this article[3].

Before proceeding to answer these questions, we need to define some terms. A COTS software component is defined as it was used in the COCOTS model [2]. This definition contains the following four parts:

1. A COTS component is sold, leased, or licensed for a fee (which includes vendor support in fixing defects if they are found).
2. The source code is unavailable.
3. The component evolves over time as the vendor provides periodic releases of the product (upgrades) containing fixes and new or enhanced functionality.
4. Any given version of a COTS component will reach eventual obsolescence or end of life in which it will no longer be supported by the vendor.

All four parts of this definition have major implications for the added costs of maintaining COTS-intensive systems (compared to comparable custom-developed systems).

At any given time for any given component, there is a choice between upgrading to the next version or doing nothing. There are risks inherent in either strategy. If the first choice is made to upgrade to a new version, there may be unintended interactions with other components, there may be defects introduced as well as unneeded functionality. These types of impacts are discussed in more detail in this article.

If the second choice is made to do nothing, the component will eventually reach end-of-life and will no longer be supported by the vendor. If a problem with the component surfaces at this point in time, the vendor will not fix it and the system maintenance team cannot do much because they do not have access to the source code.

Also, before proceeding, it is important to clarify the type of projects the interviewed managers were involved in. The Software Engineering Institute (SEI) has made a distinction between *COTS-solution* and *COTS-intensive* systems [3]. COTS-solution systems are the typical business or standard information technology systems that are comprised of large application COTS products. Examples include Enterprise Resource Planning applications, human resource, and financial systems. The major COTS component is essentially the system. It provides a user interface, has its own architecture, and has internal business logic that must be followed to be used. On the other hand, COTS-intensive systems are comprised of many COTS components. In these systems, no single component is *king*. There may be many components that handle user interface, data transmission and storage, and data manipulation and transformation. These components interact with each other through custom-developed *glue code* using vendor-provided application program interfaces and with custom-developed application code. The business logic is spread across components and is guided by the way the components are used.

The systems that predominantly made up our sample are mission-critical systems with high reliability and performance requirements and would be classified as COTS-intensive. A number of our projects were air-traffic control systems; we also had ground control systems for missile launches and two ground control systems for satellites. In addition to the high performance and reliability requirements, these systems typically had a large amount of custom application code along with a large number of COTS products (between 10 and 50 was typical).

Two additional points are worth bringing up before discussing the specific sources of added costs for these COTS-intensive systems. We deliberately chose the term *maintenance* rather than the more commonly used term *sustainment* because,

like hardware, a COTS-intensive system will, in effect, degrade without dollars and effort spent to manage the impact of multiple components evolving over time. This is the central thesis of this article and is the source of additional cost for these systems.

We do not want to leave the impression that we are against the use of COTS components. Given the complexity of many of today's systems, total custom development is no longer feasible. In addition, the use of COTS components allows system developers to take advantage of the best that the marketplace has to offer and removes the unnecessary *reinvention of the wheel* seen prior to the widespread use of COTS components. Our objective is to help people anticipate and manage the added sources of costs in maintaining COTS-intensive systems.

## Major Sources of Added Costs in Maintaining Systems With COTS Software Components

This section discusses the factors that were found to impact the cost of maintaining COTS-intensive systems. Each of the following factors is compared to custom developed systems.

### *Licensing*

The most obvious additional cost burden is component licensing fees. Fees can range from a one-time fee to yearly renewal. The license may be enterprise-wide, site-specific, or per seat (one computer). With one exception, licensing fees did not cause concern among the project members interviewed, presumably because this was an expected, known life-cycle cost. The one exception occurred for a COTS-solution system that was used on a pilot basis at one location. Following a successful pilot, the decision was made to deploy the system worldwide which would entail hundreds of sites. Much to the surprise of the project manager, the per site fee was increased by the vendor. At the time of the interview, he indicated that he assumed that they would get a quantity discount. The price per copy was actually going up. The increase in price was so great that he was seriously considering starting over, this time writing the system themselves from scratch. There are no comparable fees for custom-developed systems.

There is effort required in tracking licensing requirements to ensure that renewals are paid. With different types of licensing and support agreements across different COTS components and vendors, this tracking can become an administrative burden. There is no comparable effort required for custom systems.

### *Evaluation of New Releases*

A major source of cost stems from COTS component volatility.

> Volatility in this case means the frequency with which vendors release new versions of their products and the significance of the changes in those new versions, i.e. minor upgrades versus major new releases. [4]

In contrast to custom-developed code, a COTS software component is controlled by the vendor. The timing and content of releases is at the discretion of the vendor. Major effort may be required to evaluate and understand the implications of upgrading to a new component or perhaps switching to a whole new product entirely.

> *"Evaluation activities require a test bed that can replicate all deployed system configurations of hardware and software."*

COTS software component evaluation addresses the following questions:

1. Are there interactions with other parts of the system?
2. Are there any performance impacts on the system as a whole?
3. Will we need to rewrite *glue code* or application code?
4. Are there any impacts on any custom code?
5. Are there new features that need to be disabled?
6. If there are multiple hardware configurations in the field, can we be sure there are no unintended interactions for any of these?
7. Should we continually upgrade as new versions appear or should we only upgrade for a critical fix?

Evaluation activities require a test bed that can replicate all deployed system configurations of hardware and software. For safety-critical systems, the amount of analysis can be large even though the ultimate decision may be to do nothing. As one person stated, *even when we don't change a version, there is a lot of analysis required. It can be difficult to verify implications with a black box*. The need for this ongoing black-box evaluation is unique to systems with COTS components.

### *Defect Hunting*

Defects appear to be more problematic for COTS-intensive systems than with custom code. After documenting and confirming the existence of a defect, the next step is finding the source within the system. Projects reported that it can be much more difficult with a COTS-based system to pinpoint the source of a problem. It can be difficult to know whether a defect is coming from a COTS component or from other custom developed code. We heard of finger-pointing situations in which a defect was in a COTS product, but the vendor was unable to replicate it because they did not have the same hardware configuration. (Incidentally, this is a problem that may occur during development as well as at any point in the life cycle.) All of this can take time and effort, translating into additional costs.

With a custom system, one can see inside the box. Debugging can follow the path through the code without running into component boundaries. This eliminates finger pointing.

### *Vendor Support*

Vendor support is often used in maintenance to fix defects quickly, provide assistance with the latest product upgrades, or make adjustments to the COTS component in the presence of other product upgrades. The support may range from 24/7 call service to dedicated on-site staffing. If a defect is found and it looks like the source is a COTS component, it is the vendor who must fix the problem (provided the vendor agrees their product has the problem – as noted above, this resolution can take a lot of time and effort). A variety of contractual mechanisms can be in place to guarantee 24/7 support and immediate fixes (this, too, can be a quagmire if the support is unsatisfactory). If the latest release of a COTS software component has new features or interfaces, a vendor's support may be required to integrate a component into the current system. This support may include some tailoring by the vendor to get their component to cooperate with the existing system architecture. Finally, if a vendor has gone out of business, support may be unavailable. Risk-mitigation strategies are discussed later that address this situation.

### *Upgrade Ripple Effect*

After a new version of a COTS component has been evaluated, the installation of

the component into the system may have a ripple effect. Due to the new, additional functionality in a component, the system may require changes to custom code, glue code between components, or tailoring of other COTS components. In custom-developed code maintenance, only the fixes and enhancements that are needed are implemented, thus minimizing (but not eliminating) ripple effects.

### *Hardware Upgrades*

People found that upgrades to new software components sometimes required upgrades to new hardware as well. One person noted that vendors were constantly driven to add functionality, putting more demands on the hardware. They have not been able to upgrade the hardware as quickly as they would like.

In a comparable custom maintenance upgrade, hardware performance is considered as part of the upgrade activity. With only the required features implemented, minimal impact to hardware performance can be preserved.

### *Disabling New Features*

There may be new features that need to be disabled for security or performance reasons. The added cost is in the form of additional tailoring of the COTS component. This may require discovering how to disable new features or custom code written to hide or disable the new features. Disabling a feature is not characteristic of custom systems.

### *Early Maintenance*

Because COTS components continue to evolve in the marketplace, it is possible that upgrades may begin before the system is deployed, particularly if the development spans several years. If the components are not upgraded, it is possible that much of the system may have reached *end of life* before the system is even delivered. This was the case according to one of the project managers interviewed; this system had an application base totaling more than one million lines of custom code plus a total of 45 COTS components. Almost half of these components were obsolete by the time the system was deployed.

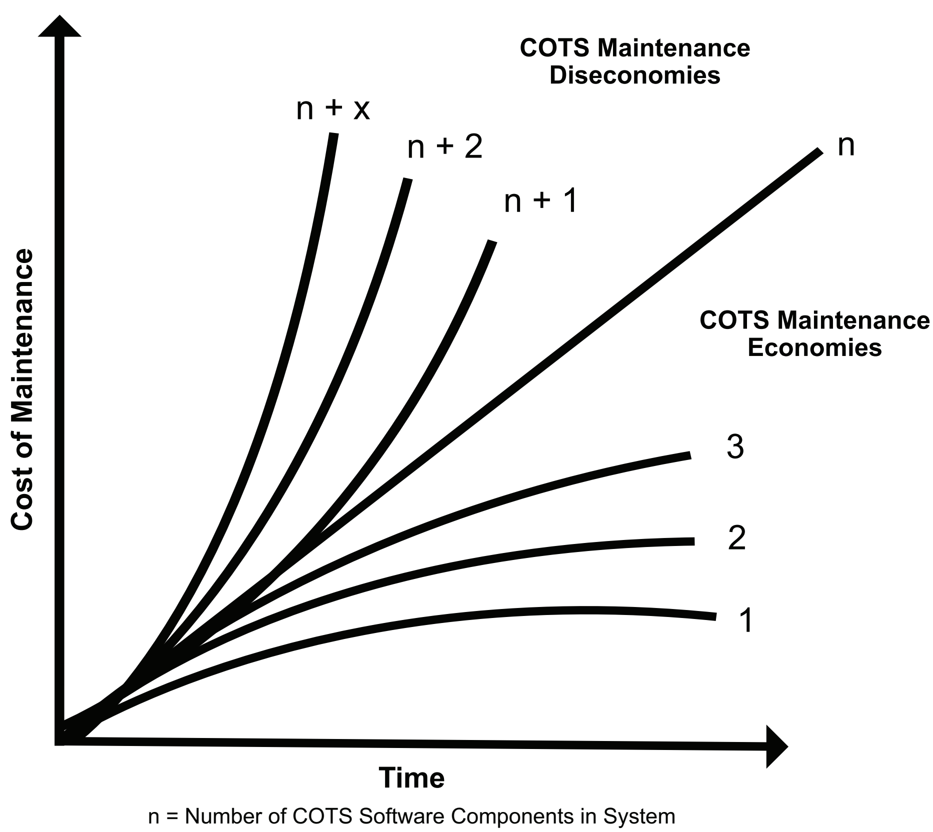
### *Market Watch*

Because COTS vendors can go out of business, a number of those interviewed suggested that a *market watch* be established as a risk mitigation strategy to handle such an event. If a vendor goes out of business, either the component source code or a different component can be purchased. With custom-developed systems, this activity is not required.

### *Continuous Funding*

Another difference between a COTS-based and a custom system is that the systems with COTS components require a more stable funding base. When budgets get tight, funding for maintenance is often sacrificed. With a custom system, enhancement can be delayed until funding is obtained. The consequences of delaying funding with a COTS-based system is that licenses may lapse, bug fixes and upgrades become unavailable, or vendors go out of business with no resources to exercise the risk mitigation identified in a market watch.

Figure 1: *COTS Maintenance Economies Versus Diseconomies*




### *Number of Components Versus Maintenance Costs*

One consistent comment we heard is that the number of COTS components in a system has a strong impact on maintenance costs. A model adapted from one proposed by Chris Abts, called COTS-Life Span Model (LIMO) [5], attempts to explain this phenomenon. The model, depicted in Figure 1, shows two regions divided into maintenance economies (overtime costs go down) and maintenance diseconomies (overtime costs go up). As explained in COTS-LIMO, maintenance costs for a single COTS component go down over time as the experience gained by system maintainers increases, thus improving productivity. The increase in productivity can outpace the increased effort required to maintain the system as the COTS products mature and evolve in divergent directions. However, there is a break-even point with the number of installed COTS software components, (n), where maintenance costs increase disproportionately to the number of COTS products, regardless of the efficiencies gained.

Factors that contribute to the COTS maintenance diseconomies in Figure 1 are those that are discussed in this article. For instance, issues raised with COTS licensing is much more complex with more components. A COTS-intensive system presents multiple licensing strategies, different renewal periods, and different license cost structures. People reported that this can become an administrative nightmare.

Evaluating the impact of upgrades is considerably more burdensome if there are a lot of components (greater than n). The number of possible interactions between components increases exponentially as the number of components increases. When trying to hunt down defects, the complex interactions of many components make the task even more difficult. Configuration management becomes more complex when many compo-

nents and configurations exist in a system. The possibility of a ripple effect is higher with the impact of component upgrades. There are more unwanted features with more components. The market watch becomes a large-scale activity.

The idea of this model was verified when we kept hearing that the complexity of maintaining a COTS-based system increases dramatically as the number of different COTS components increases. There is much more potential for interaction as well as more potential upgrades that have to be examined.

## Three Risk-Mitigation Strategies to Deal With the Challenges of Maintaining COTS-Intensive Systems

This article has discussed sources of additional cost in maintaining COTS-intensive systems. Across the projects interviewed, three strategies for dealing with COTS volatility were observed. Each of these strategies is discussed next.

### *Revert Back to Source Code*

Several of the projects interviewed opted to maintain one or more *critical* COTS components themselves. In one case, the product (an operating system) was allowed to reach end of life and the project purchased the source code from the vendor. From that point on, they no longer had vendor support but were able to make fixes themselves. This decision was made because it avoided the necessity for hardware upgrades. It removed the risk of being unable to fix future problems. Alternatively, several other projects replaced critical COTS components with their own custom-developed software.

This strategy places control for fixing problems back in the hands of the maintenance organization. A downside is the additional expense (purchasing the source code or developing it from scratch). In the case given dealing with the operating system, this strategy was part of a larger strategy to freeze the hardware configuration, much of which was special purpose, for a period of time until the next generation system could be deployed.

### *Divide and Conquer*

This strategy divides the COTS software components into two categories: non-critical and critical. The non-critical COTS components are not upgraded. Resources are focused on the set of critical components. For these components, market watch and evaluation activities occurred and the decision to upgrade was made individually for each critical component.

This strategy is driven by the need to balance the ongoing costs required for maintaining a COTS-intensive system with limited resources. The upside of this strategy is that it saves money by ignoring a subset of components. The downside of this strategy is that a portion of the system remains stagnant and unsupported.

### *Design for Change*

The third strategy uses *information hiding* in the form of wrappers to protect the system from unintended negative impacts of multiple component upgrades. One interviewee said when describing this strategy that they wanted to be able to replace a product without damage to the rest of the system. As an example, they had a wrapper around the database. It could be a flat file or relational database – the custom application didn't care. This strategy requires more thought and effort up front. The project in our sample that used this strategy had a strong project sponsor right from the beginning who argued successfully for additional resources to design for change from the beginning. This was a project that was planned for a long life with safety-critical requirements.

> *"The sources of added costs discussed ... were identified through anecdotal evidence obtained from interviews. The next step is to quantify these sources and the parameters that impact each. We are looking for opportunities to continue this investigation."*

The advantages of this strategy are clear: There is much more assurance against unintended ripple effects from upgrades or even product replacement with a product from another vendor. The disadvantage is the necessity for resources early in system development when the typical focus is on getting the system deployed rather than worrying a great deal about the life-cycle consequences of decisions.

## Concluding Remarks

This article has discussed the sources of additional costs required to maintain COTS-intensive systems. As noted in the introduction, we do not want to leave the impression that we are against the use of COTS components. One of the people interviewed expressed the view that the continual evolution and maturation of COTS components is, in fact, one of the real positives of using commercial components in a system.

It is the authors' objective to help people understand some of the added sources of costs in maintaining a COTS-intensive system, particularly by bringing attention to areas that may not have been anticipated. In particular, projects should understand the life-cycle implications of integrating a large number of COTS components. More thought should be given early to the impact of upgrades on the entire system, the reliance on vendors to fix problems, and the strategies that will be used in dealing with multiple products, each evolving at the discretion of the vendor. These concerns become especially problematic with high assurance, high performance systems.

The sources of added costs discussed in this article were identified through anecdotal evidence obtained from interviews. The next step is to quantify these sources and the parameters that impact each. We are looking for opportunities to continue this investigation.◆

## References


1. Reifer, D.J., V.R. Basili, B.W. Boehm, and B. Clark. "COTS-Based Systems – Twelve Lessons Learned about Maintenance." COTS-Based Software Systems, Third International Conference, ICCBSS 2004, Redondo Beach, CA.
2. Center for Systems and Software Engineering. COCOTS <http://sunset.usc.edu/research/COCOTS/index.html>.
3. Oberndorf, Tricia, Lisa Brownsword, and Carole Sledge. "An Activity Framework for COTS-Based Systems." Carnegie Mellon University (CMU)/SEI-2000-TR-010. Pittsburgh: SEI, CMU, 2000.
4. Abts, Chris. "COTS-Based Systems and Make vs. Buy Decisions: The Emerging Picture." International Workshop on Reuse Economics, Austin, TX. 16 Apr. 2002 <www.

## Coming Events

**July 9-11**
*CBSE 2007 10th International ACM SIGSOFT Symposium on Component – Based Software Engineering*
Boston, MA
www.csse.monash.edu.au/~hws/CBSE10

**July 9-11**
*SEDE 2007*
*16th International Conference on Software Engineering and Data Engineering*
Las Vegas, NV
http://sede07.cs.uh.edu

**July 9-11**
*SEKE 2007*
*19th International Conference on Software Engineering and Knowledge Engineering*
Boston, MA
www.ksi.edu/seke/seke07.html

**July 9-12**
*SETP 2007*
*International Conference on Software Engineering Theory and Practice*
Orlando, FL
www.promoteresearch.org/2007/setp/index.html

**July 15-18**
*SCSC 2007 Summer Computer Simulation Conference*
San Diego, CA
www.sce.carleton.ca/faculty/wainer/SCSC07/SCSC'07.htm

**2008**

*Systems and Software Technology Conference*
www.sstc-online.org

***Coming Events:*** **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

favaro.net/john/icsr7/papers.htm>.
5. Abts, Chris. "COTS-Based Systems Functional Density – A Heuristic for Better CBS Design." Proc. First International Conference on COTS-Based Software Systems, Feb 2002.

## Notes

1. This model, named Constructive COTS Model (COCOTS), is one of the COCOMO family of models.
2. The interviews and model calibration were sponsored by the Federal Aviation Administration's (FAA) Software Engineering Resource Center. The interviews were conducted by Dr. Chris Abts and Dr. Betsy Clark.
3. For a discussion of lessons learned in maintaining COTS-intensive systems, see Reifer, et al. [1].

## About the Authors

**Betsy Clark, Ph.D.,** has been involved in the practical application of measurement for predicting, controlling and improving software process and product quality since 1979. She is the president of Software Metrics, Inc., a Virginia-based consulting company she co-founded in 1983. Clark is a primary contributor to *Practical Software Measurement: A Guide to Objective Program Insight*. She was also a principle contributor to the SEI's core measures. She has contributed to numerous studies of software best practices for the DoD and FAA. Clark is a research associate at the Center for Systems and Software Engineering at the University of Southern California. She worked with Barry Boehm and Chris Abts to develop and calibrate a cost-estimation model for COTS-intensive systems under sponsorship of the FAA. She is currently supporting the U.S. Customs and Border Protection's Cargo Systems Program Office, working to implement performance measures within the framework of the Office of Management and Budget's Performance Reference Model. Clark has a bachelor of arts in psychology with distinction from Stanford University and a doctorate in cognitive psychology from the University of California, Berkeley. She is also an accomplished equestrian, having earned a Gold Medal from the United States Dressage Federation.

**Software Metrics, Inc.**
**4345 High Ridge RD**
**Haymarket, VA 20169**
**Phone: (703) 754-0115**
**Fax: (703) 754-3446**
**E-mail: betsy@software-metrics.com**

**Brad Clark, Ph.D.,** is an independent consultant in the area of software measurement with 12 years experience in software development and management best practices. He is vice-president of Software Metrics, Inc., and specializes in the area of software cost and schedule risk analysis. Clark is a research associate with the Center for Systems and Software Engineering at the University of Southern California. He has co-authored the book *Software Cost Estimation With COCOMO II* with Barry Boehm and others. Clark helped define the COCOMO II model, collected and analyzed data, and calibrated the model. He has a bachelor's degree in computer and information science from the University of Florida, a master's degree in software engineering, and a doctorate in computer science from the University of Southern California. He is a former Navy A-6 pilot.

**Software Metrics, Inc.**
**4345 High Ridge RD**
**Haymarket, VA 20169**
**Phone: (703) 754-0115**
**Fax: (703) 754-3446**
**E-mail: brad@software-metrics.com**

# Issues to Consider Before Acquiring COTS

Dr. David A. Cook
*The AEgis Technologies Group, Inc.*

*In today's software and acquisition environment, the decision to use commercial off-the-shelf (COTS) and government off-the-shelf (GOTS) is not only necessary from a schedule and cost point of view, but often is mandated. However, there are many factors that influence the decision to use and choose COTS/GOTS software. Readers who have a background in computer science or who have taken some formal software engineering code development classes are typically familiar with the effects, but many engineers who acquire COTS do not have this background. This article discusses some basic but often-neglected factors affecting COTS selection and use.*

Back in the early days of computing, COTS use was commonplace. However, instead of integrating with other programs, the COTS packages ran independently. It was commonplace to lease or buy accounting or other similar software from your computer vendor. Locally written software seldom, if ever, interfaced with COTS. Programs tended to be relatively small, and interaction between programs consisted primarily of the output of one program being fed in as input to other programs.

As computing power increased, and computer use proliferated, more and more programs interacted with other programs, and all interacting programs ran concurrently. As this occurred, software manufacturers filled the need for common programs by devising programs that were not made to run independently, but instead provided partial solutions to the overall problem. The intent was that you could purchase a product that would meet one or more of your requirements and simply plug in this product.

COTS was, at one time, envisioned as a massive time and money saver. You would go shopping at a software warehouse and select COTS systems as needed. It would seamlessly interact with your organic, homegrown program and also with any and all other COTS products that you selected. In a sense, COTS would operate like the ubiquitous Universal Serial Bus (USB) *plug and play* devices that have become so common.

Unfortunately, this promise of *plug and play* software was not as effortless as *plug and play* was for hardware. In hardware, there are very specific standards that specify how the interface to the computer must occur. This allows a hardware creator to design a product that is specific in its purpose (for example, a 256 Meg flash drive) but general in its interface (so that when inserted, drivers automatically load with any disk insertion or user interaction). In hardware, there are general device drivers that have extremely specific interface standards. Plug in a USB drive from almost any manufacturer and the device will work without any specific *tailoring* of the device to the hardware.

COTS, on the other hand, typically do not have specific standards. COTS requires the user to adapt his/her environment to use the COTS. Whereas the *plug-and-play* hardware has a known and generalized standard that all users are willing to adapt to, COTS requires each user to adapt their individual software to interface with the COTS. Each user typically has specific and unique needs, yet they want to interact with general-purpose COTS. This usually causes problems in that the COTS does not meet all of the user needs, and in fact might contain functionality that the user does not want. Explaining how COTS interacts with other applications can be better understood by examining two relatively common software engineering concepts: coupling and cohesion. Coupling and cohesion, which are relatively basic terms in computer science, can be applied to COTS to help determine COTS quality and also to determine how easy it will be to integrate COTS with locally developed code.

## Coupling

Coupling refers to how programs interact. There is a range of how multiple modules interact (see Table 1 for the different types of coupling). In the best

Table 1: *Types of Coupling, Ranked From Best to Worst*

| | |
|---|---|
| **Message Coupling (low coupling, and the best type)** | This is the loosest type of coupling. Modules are not dependent on each other; instead, they both use a public interface to pass messages between them, such as an object-oriented message. |
| **Data Coupling** | Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data that are shared (e.g. passing an integer to a function that computes a square root). |
| **Stamp Coupling (data-structured coupling)** | Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g. passing a whole record to a function which only needs one field of it). This may lead to changing the way a module reads a record because a field that the module does not need has been modified. |
| **Control Coupling** | Control coupling is one module controlling the logic of another by passing it information on what to do (e.g. passing a what-to-do flag). |
| **External Coupling** | External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface (e.g. you have no control over the interface). |
| **Common Coupling** | Common coupling is when two modules share the same global data (e.g. a global variable). Changing the shared resource implies changing all the modules using it. |
| **Content Coupling (highest coupling, and the worst type)** | Content coupling is when one module modifies or relies on the internal workings of another module (e.g. accessing local data of another module). |

Source: <http://en.wikipedia.org/wiki/Coupling_(computer_science)>

scenario, modules each fulfill a specific purpose and no direct interaction is required. This lack of coupling is perfect in that the actions of one program or module have little or no effect on other programs. However, this type of scenario is reminiscent of olden days when programs ran sequentially, rather than parallel. Today, COTS almost always requires interactions with other concurrently running programs.

Prior to C++, the best type of coupling was referred to as *data coupling*. In data coupling, all interaction is defined by parameter calls. This type of coupling is simple and the least likely to cause a ripple effect – that is, when a change to the logic of one module causes undesirable effects to other modules. The advent of commonly used object-oriented languages has led to a new (lower, and therefore better) level of coupling, referred to as *message coupling*[1]. In a well-designed system, low coupling gives COTS the ability of *plug and play* in terms of a standard interface. With only *message coupling* or *data coupling*, it should be relatively easy to remove one COTS module and replace it with a similar COTS module that has the same interface.

Back in the 1970s, the first car I owned – a 1973 Chevrolet Impala – had an AM/FM radio, but I wanted to replace it with an AM/FM/Cassette radio. After disassembling the dashboard (no small feat), I removed the radio to find out that it had six black wires, one white wire, and one green wire. The new radio had seven black wires and a white/green twisted pair. I had no idea how to cleanly replace one radio with the other. A more recent car, bought a few years ago, had a standardized harness plug. When I asked to replace the standard AM/FM/Cassette/CD with a six-CD radio, it took about 10 minutes – basically just unplug the old radio and plug in the new. This is the kind of interfacing you want with your software: easy to uninstall the old, easy to install the new. When your current COTS supplier goes out of business or no longer supports your version, it should be easy to replace.

It is not important, of course, to determine exactly what the level of coupling is – it is more important to keep it low[2]. Coupling is directly related to information hiding, along with data abstraction and data design. The moral of coupling is to *really* design systems that interact (or might interact) with COTS. Use interface control documents. Have an architectural design document. Review data requirements with the user, have a data dictionary and data design document, and make the COTS interfaces as simple and straightforward as possible. Coupling should be kept as *low* as possible.

Table 2: *Types of Cohesion, Ranked From Best to Worst*

| | |
|---|---|
| **Functional Cohesion (high, and the best type of cohesion)** | Functional cohesion is when parts of a module are grouped because they all contribute to a single, well-defined task of the module (e.g. calculating the sine of an angle). |
| **Sequential Cohesion** | Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data). |
| **Communicational Cohesion** | Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on one record of information). |
| **Procedural Cohesion** | Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file). |
| **Temporal Cohesion** | Temporal cohesion is when parts of a module are grouped by when they are processed – the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user). |
| **Logical Cohesion** | Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all input/output handling routines). |
| **Coincidental Cohesion (low and the worst type of cohesion)** | Coincidental cohesion is when parts of a module are grouped arbitrarily (at random); the parts have no significant relationship (e.g. a file of frequently used functions). |

Source: <http://en.wikipedia.org/wiki/Cohesion_(computer_science)>

## Cohesion

Cohesion refers to the measure of how COTS performs a single task. It is a measure of the *stickiness* of the module. A good module contains only resources that accomplish single or similar tasks. The parts of the module are all closely interrelated and therefore stick together. Table 2 explains the types of cohesion.

Many studies have shown that coincidental cohesion (such as one routine that would *calculate tax rate and compute Celsius to Fahrenheit*) and logical cohesion (such as *open all input files*) are extremely inferior to other types of cohesion[3]. Combining many functions into one module makes cohesion low and contributes to high error rates and increased debugging, testing, and integrating time. In addition, relatively small maintenance fixes tend to have ripple effects that require testing and debugging of many routines that are not logically related. Good developers know that an application that has many relatively simple modules is easier to develop and test. In fact, this is one of the basic concepts of object-oriented programming.

For most computer users, the programs we use have a single purpose. Most computer users have learned that it makes sense to have a single word processor, a single spreadsheet program, etc. While they might all come from the same software developer and be designed to interact together, they are still separate programs. Back in the 1980s, there was a push to deliver *all-in-one* software that combined lots of functionality. It was clumsy to use, performed poorly, and was widely hated by most users. Instead, single-purpose applications (which had high cohesion) that had well-defined *cut and paste* and hyperlink interfaces (low coupling) tend to be easier to use, to be more robust, and to include more functionality. COTS works the same way. A single-purpose program typically works better and is almost always easier to integrate.

Cohesion is directly related to modularity (whereas coupling was directly related to information hiding). In a well-modularized program, routines are reasonably small and perform one action. In a poorly modularized program, routines grow very large – sometimes thousands of lines. Good software engineers know that it is easier to write, test, and

then integrate lots of small routines rather than try to write and debug monolithic modules of several thousand lines.

Whereas it is important to keep coupling low, cohesion should be kept high. To design for high cohesion, architectural and module design documents are needed. Good requirements are a must, as are traceability matrixes. A traceability matrix allows you to map the requirements directly to the design and module and encourages you to have small modules that meet a small number of requirements.

Again, there are metrics to measure cohesion, just as there are metrics to measure coupling. However, it is not as important to measure coupling or cohesion; it is more important to try and keep coupling low and cohesion high. With COTS, you want to insert off-the-shelf functionality that provides both single and well-defined functionality.

## COTS Testing and Requirements

It is obvious in hindsight, but how do you *know* that the COTS products that you are buying will meet your needs? Certainly you cannot believe the marketing hype, and because COTS products are developed for many users, your application might have unique COTS needs that have never been tested for or used in other applications. The solution is to have a well thought-out test plan that makes sure that your unique needs are tested. This test plan needs to be developed *before* you acquire COTS. And, to develop a good test plan, you need good requirements. The bottom line is that if you acquire COTS before you have high-quality requirements that have been validated by the user, you are likely to end up with COTS that do not meet all of your needs.

Of course, maybe there are no COTS products that meet all of your needs, anyway. This is a common occurrence. There are several solutions: Either modify the COTS or develop so-called *glue* code that holds the COTS together.

I highly recommend against attempting to modify COTS. Such attempts typically lead to increased cost and lengthy testing[4]. Instead, consider either devising a manual work-around or implement a limited amount of *glue* code. However, keep in mind that if the *glue* code is complex and hard to maintain, it might overcome the cost and time savings of using COTS.

## Version, Speed, and Licenses

Other issues affect the overall cost of a COTS system. Systems with a lot of COTS components have a problem with versioning, both with the versioning of the COTS and with the underlying operating system. When COTS is updated by the vendor, it requires careful regression testing to make sure that the newer versions of COTS continue to meet the exact *form, fit,* and *function* of the previous version. In addition, care must be taken that newer versions, as they are released, do not add unwanted functionality (and vulnerabilities).

A second issue is that in addition to successive versions of newer COTS, each COTS package might want specific versions or patches to the operating system. Each might want a separate version of the Java Machine installed. As updates to both the operating system and system libraries are installed (in some cases automatically), how do you know that the COTS will continue to work? Again, a test team must be continually testing and checking to make sure that updates to the operating system do not cause unintended side effects. It helps if the COTS has a scheduled update cycle and if pre-release versions of the upcoming COTS can be tested early.

> *"I highly recommend against attempting to modify COTS. Such attempts typically lead to increased cost and lengthy testing."*

A third issue with versioning becomes important when you have multiple COTS running. As the matrix of interrelated COTS packages are updated, interfaces from one COTS package might interfere with other COTS interfaces. System and hardware requirements might conflict, or be vastly magnified with multiple COTS packages installed.

To address all of these issues, a test plan (and a dedicated test team) is critical. Research into the interconnection between various COTS packages is required on a continual basis. In the past, the principle was typically *get it running and lock down the entire configuration* and baseline the system (and prevent continual COTS and operating system updates). However, because of security threats, operating systems of today require continual updating to meet increasing security threats. It is no longer practical to *lock down* an environment.

Other issues with COTS involve the performance of the integrated system. In any system, it is important to test (or at least prepare for) *worst case* conditions. Unfortunately, as requirements change during development, worst case conditions sometimes change, also. When you acquire COTS for a specific need and the need changes during development, performance problems can occur. The performance/speed issues can also involve hardware. Remember that you need to develop and test the entire system for not only the average user, but also the worst-case user. Finding out that the users have inadequate hardware after development and release of the software is bad. Also, remember that hardware preparation also includes network support. It would be catastrophic to deliver a finished system only to find out that there is not enough available bandwidth necessary to support the new COTS database. Again, this is something that can be avoided by having good requirements and by having a test team testing against the requirements.

A final issue is licensing costs. For some COTS, a separate license must be acquired for each user. Do you know how many end-users will need licenses? Is there an acquisition mechanism in place to buy and distribute the licenses when you release the COTS? Proper preparation for release requires that you make plans for licenses and license distribution.

These are just a few issues that affect the overall cost of COTS. In this issue of CrossTalk, the article *Added Sources of Costs in Maintaining COTS-Intensive Systems* by Clark and Clark explores these additional costs in greater detail.

## Conclusion

COTS acquisition and integration are complex topics – this article simply touched on some of the basic issues to consider. Coupling and cohesion are relatively basic computer science concepts, but an understanding of how they relate

to good software engineering can make the job of acquiring COTS easier. From a larger perspective, basic COTS questions can be answered by basic *good* software engineering: Gather good requirements, validate the requirements, perform good design, and have a test plan. COTS probably will not solve 100 percent of your requirements, so it is important to know what you are willing to settle for. Also, understand that COTS might save you time and money, but trying to integrate it incorrectly can cost you lots of money and will not necessarily reduce the schedule time. Trying to integrate multiple COTS applications will probably magnify your potential problems; so learn from the mistakes and successes of others whenever possible. Be wary of marketing hype, and remember that finding another user with experience with your potential COTS product is one of the best resources of all.◆

## Notes

1. For a discussion of program language evolution and programming language interoperability, refer to "Evolutionary Trends of Programming Languages," by Thomas Schorsch and David Cook in CrossTalk Feb. 2003 at <www.stsc.hill.af.mil/crosstalk/2003/02/schorsch.html>.
2. See "Software Engineering, A Practitioners Approach" by Roger Pressman for a good description of coupling, cohesion, information hiding, abstraction, and modularity. You can also get pointers to online sources of information by checking out *coupling* and *cohesion* on Wikipedia.
3. For example, see E. Yourdon, and L. Constantine. "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design." Prentice-Hall, Yourdon Press.
4. For an example, see "Business Process Reengineering/Software Modifications" OIG/97E-10 – "Evaluation of Best Practices for Developing and Implementing Integrated Financial Management System From the U.S." by the Nuclear Regulatory Commission, available at <www.nrc.gov/reading-rm/doc-collections/insp-gen/1997/97e-10.html>.

## About the Author

**David A. Cook, Ph.D.,** is a senior research scientist at AEgis Technologies, working as a verification, validation, and accreditation agent in the modeling and simulations area. He has more than 30 years experience in software development and management. Cook was associate professor and department research director at the U.S. Air Force Academy and former deputy department head of the software professional development program at Air Force Institute of Technology. He was a Software Technology Support Center consultant for six years. Cook has a doctorate in computer science from Texas A&M University.

**The AEgis Technologies Group, Inc.**
**6565 Americas Parkway NE**
**Albuquerque, NM 87110**
**Phone: (505) 881-1003**
**Fax: (505) 881-5003**
**E-mail: dcook@aegistg.com**

# Lean AISF: Applying COTS to System Integration Facilities

Harold Lowery
*Warner Robins Air Logistics Center*

*Lean Avionics Integration Support Facility (AISF) is an initiative to introduce Lean concepts and methods to the F-15 AISF. Our strategy includes the use of commercial off-the-shelf (COTS) and open source software where appropriate. In this article, the author briefly describes the AISF and then discusses several examples of using COTS to reduce maintenance costs and improve performance.*

Modern weapons are complex, high-performance systems. Much of the performance of a modern weapon system, as well as its complexity, derives from the software executing on computers embedded within it. It should come as no surprise that the engineering facilities used to develop and maintain these weapon systems are themselves complex systems that require considerable resources to operate and maintain. The application of *Lean* concepts enables significant cost reductions in the maintenance of system integration labs through the use of COTS items where appropriate. This article describes one such facility, the AISF located at Robins Air Force Base, and discusses several examples of how new technology impacts it.

## Fighter AISF

In order to discuss Lean AISF, we first must discuss the Fighter AISF history.

### *History*

The Fighter AISF is used to develop and maintain Operational Flight Program (OFP) software, primarily for the F-15 and other air combat platforms. The AISF achieved initial operation in the early 1980s and has been through several technology refresh cycles since then. The AISF includes a number of system integration benches. These benches are closed loop, hardware-in-the-loop systems consisting of avionics hardware, signal processing hardware and software, and simulation software. The OFPs execute in actual aircraft avionics with the airframe and operating environment simulated. The intent is that the OFP software cannot tell the difference between flying in an aircraft and flying in the lab.

### *Principles of Lean*

In the early 1990s, researchers began discussing the concept of a *Lean* approach to manufacturing. Womack, Jones, and Ross introduced the term *Lean* when describing the Toyota Production System as part of a major study of the global automotive industry [1]. The concepts they described – focusing on the value stream and eliminating waste – have been successfully applied to manufacture and repair processes in the automotive and aerospace industries for some time. Innovative organizations are now applying *Lean* principles to their design and product development challenges. The emphasis in this domain is on eliminating waste, particularly in make-vs-buy decisions [2].

### *Application of Lean*

Our initiative has the goal of transforming the traditional AISF to a Lean AISF, by moving from obsolete hardware/software to modern systems that are based on COTS equipment, open industry standards, and open source software where appropriate. In particular, we aim to lower the cost to support the AISF by applying Lean principles to product development to eliminate waste whenever possible. The expected benefits of this transformation are reduced hardware maintenance costs for AISF hardware, easier migration of new technology into existing AISFs, and reduced development costs for new AISFs to support weapon systems currently in development.

> *"The application of Lean concepts enables significant cost reductions in the maintenance of system integration labs through the use of COTS items where appropriate."*

Meeting the stringent real-time constraints of simulating a fighter requires significant computing horsepower. The first, second, and third generations of the AISF, like all system integration facilities built during the 1980s and 1990s, were based on expensive minicomputer hardware running proprietary operating systems and software development toolsets. In addition, a large investment in custom-designed hardware and software was necessary to meet the system's requirements, using the then-available technology. In implementing the fourth generation AISF, our aim is to eliminate waste, especially in the make-vs-buy decisions that so strongly drive life-cycle costs.

## Example 1: Simulation Computers

For years, we have used minicomputers from a major simulation vendor to host our real-time simulation software. These machines have been true workhorses for us, but with the passage of time there were several reasons to move to newer technology. First, since these machines are based on the vendor's proprietary hardware, we have supported them via vendor maintenance contracts. This approach gave us superb support but was a strain on the budget. Second, as technology has advanced, our options for upgrading these computers grew limited. For example, the largest hard drives that they could accommodate are two gigabytes: This was great when the computers were new in 1991 but rather constrained some 15 years later.

We conducted a trade study to evaluate three alternatives. First, we could migrate from our existing simulation computers along the vendor's upgrade path to their next generation product. A significant feature of this alternative is the move to the open source Red Hat Linux operating system with the RedHawk real-time kernel. Second, we could build our own simulation hosts using COTS hardware running Linux. Third, we could expand the search space to the proprietary simulation products of other commercial vendors.

Alternatives one and two both used standard Intel-based servers running a version of Linux. The tremendous growth of the Internet had driven massive industry investment in servers, lowering the unit price of raw processing power. Alternative

one also included the vendor's proprietary hardware and software to provide a system optimized for real-time processing, albeit at a significantly higher price.

At first it was tempting to believe we could assemble a solution in-house by using off-the-shelf hardware (which we would buy strictly on price) and installing Linux. However, to re-host our legacy software to such a platform would require specific real-time capabilities – capabilities we would have to develop from scratch. As we began to tally up the engineering development costs, it became clear that cheap hardware could be too expensive.

Our trade study evaluated the alternatives on the basis of the following: 1) real-time capability, 2) supportability (over a nominal 10-year design life), 3) purchase costs, and 4) transition costs (including costs to re-engineer existing simulation software). Alternative one was the clear winner. The vendor's solution provided us an upgrade path where we could port our large legacy code base with minimal effort relative to other approaches. Although we could have bought equivalent hardware for half the price from other sources, the ability to quickly port our large legacy code base was a value proposition that surpassed the other alternatives.

### *Lesson Learned*

Hardware may be cheap, but software engineers are expensive. When dealing with legacy systems, we found that the most cost-effective approach is generally the one that minimizes the software re-host effort.

## Example 2: Bus Interface Cards

The H009 multiplex bus was an early fore-runner of the Military Standard (MIL-STD)-1553B data bus that has become standard in military and even commercial aircraft. Since H009 was never as widely adopted as 1553B, there have always been relatively few suppliers of this hardware.

From the early days of the AISF, we made significant investments in designing, building, and maintaining custom H009 interface cards for the AISF. Our most recent implementation was designed in the early 1990s and had become unsupportable due both to obsolescence and personnel turnover. We had entered the H009 business simply because at the time we felt there were no viable commercial alternatives. In recent years, the engineering expertise to support this very specialized design across a small installed base (about 12 units, total) had eroded significantly. We had a strong desire to stop supporting in-house custom solutions, and by 2005 several vendors were offering H009 products.

As we investigated them, it quickly became clear that none would operate in our system without a significant rewrite of our existing software. In our third-generation hardware design, the software engineers had requested a number of features they thought would be needed. Over the last dozen years we had learned that some of those features were seldom, if ever, used – a form of waste. In effect, our board had been designed with some capabilities that were unnecessary and with others that were perhaps better done in software. In order to use the available COTS hardware, we would have to migrate some of the functionality of our custom hardware into an enhanced version of our software.

> *"Hardware may be cheap, but software engineers are expensive. When dealing with legacy systems, we found that the most cost-effective approach is generally the one that minimizes the software rehost effort."*

We had to trade off the costs of implementing a new custom hardware design and then supporting it for a number of years versus the one-time cost to modify the legacy interface software to accommodate the feature set offered by off-the-shelf solutions. Another factor we considered in our analysis was available support for the COTS equipment. Fortunately, the F-15 is gradually migrating away from the H009 bus to the much better supported MIL-STD-1553B bus. By provisioning the proper number of spare cards, we expect to support the H009 bus for as long as it remains in use on the aircraft.

### *Lesson Learned*

A one-time investment of engineering dollars can be cost effective if it allows the use of COTS equipment and eliminates the engineering effort required to design and support an in-house solution over a period of years.

## Example 3

In the first generation AISF, circa early 1980s, we used real aircraft control panels in our cockpit mock-ups. Although these gave the user a realistic experience in the lab, there were several drawbacks with them. Aircraft hardware is expensive to obtain, difficult to maintain, and has to be interfaced to the simulation computers using custom hardware. In our second-generation designs (late 1980s), we began experimenting with touch-screen equipped PCs as replacements for aircraft control panels. This approach eliminated aircraft hardware while still allowing us sufficient realism for the purposes of OFP development. However, implementing that approach required developing the software to display buttons, switches, etc., and to respond to the user as he or she activated these simulated controls. At the time, this meant a significant investment in custom software development.

Fast forward to 2006. Our original touch screen PC hardware had been replaced several times, but the software had been modified only slightly over the years and was in definite need of a major overhaul. But now we had options. The market for PC graphics software has greatly increased and several vendors offered promising products – the promise being that re-implementing our existing applications would be as easy as drawing the panels using the vendor's graphical editors. The old-timers among the technical staff were skeptical that it could be that easy, while the younger engineers were eager to try out new toys.

We evaluated various products and then made the investment. By using the vendor's tool, a trained engineer could prototype a control panel in a fraction of the time it would have taken with hand-crafted code.

However, what the tool saved us in creating panels it took back in time to integrate them when new hardware arrived. One significant problem involved a Linux driver that assumed a specific hardware configuration different from what we had purchased. In the end, a senior engineer rewrote the driver so that all the pieces would play together.

### *Lesson Learned*

The young engineers were right that the COTS tools would simplify the process of generating control panels. But the grey-

beards were right too. There are always integration issues, and it is precisely this point where one vendor's product meets another's that the hard work usually takes place.

## Conclusion

A Lean approach to AISF development and support would eliminate waste whenever possible. COTS products can be incorporated to great advantage if the engineering staff carefully weighs all alternatives when considering make-versus-buy decisions. In those cases where a COTS product is appropriate, it can eliminate the waste of supporting a custom solution using expensive in-house engineering talent. As always, it is important to clearly define the trade-offs and ramifications of using a COTS product.◆

## References


1. Womack, James, Daniel Jones, and Daniel Ross. The Machine That Changed the World: The Story of Lean Production. New York: Rawson Associates, 1990.
2. Kearney, A.T. "The Line on Design – How to Reduce Material Cost by Eliminating Design Waste." Electronic Datasystems Corporation, 2003.


## About the Author


**Harold Lowery** is the Director of B-Flight (F-15 AISF Support B227), 577th Software Maintenance Support Squadron. The majority of his 25 years of experience with the U.S. Air Force have been devoted to the development and support of F-15 avionics systems, with occasional excursions onto other Air Force platforms. When away from work, his five children consume the remainder of his time, energy, and patience.

**577 SMXS/FLT B**
**420 Richard Ray BLVD**
**STE 100**
**Robins AFB, GA 31098-1640**
**Phone: (478) 926-9525**
**DSN: 468-9525**
**Fax: (478) 926-9525**
**DSN: 468-2682**
**E-mail: harold.lowery @robins.af.mil**


## Web Sites

### Integration of Software-Intensive Systems (ISIS)

www.sei.cmu.edu/isis

The Software Engineering Institute ISIS initiative helps organizations successfully achieve system of systems interoperability by addressing the gaps between the net-centric vision and the status and capabilities of technologies that are being targeted to fulfill the vision, developing methods and techniques for helping organizations migrate to a network centric environment, identifying best practice for interoperability and integration, disseminating findings and guidance to the acquisition and development communities, maturing, and transitioning practical solutions into DoD organizations and the wider community.

### Ten Commandments of COTS

https://acc.dau.mil/CommunityBrowser.aspx?id=24403

Interest in COTS products requires examination both in terms of its causes and effects, and in terms of its benefits and liabilities. The Defense Acquisition University offers some observations and voices some specific concerns and criticisms. They stress that their observations are essentially cautionary, not condemnatory: Huge growth in software costs will continue, not abate, and appropriate use of commercially available products is one of the remedies that might help to acquire needed capabilities in a cost-effective manner. Where use of an existing component is both possible and feasible, it is no longer acceptable for the government to specify, build, and maintain a comparable product.

### COTS Journal Online

www.cotsjournalonline.com

Taking you into the world of the military acquisition machine, *COTS Journal* provides in depth coverage of commercially available embedded technology and its specific uses in military electronics and equipment. The subscription is free online, and archives can be accessed via the Web site without signing up just by clicking on the archives button on the menu bar.

# GL Studio Brings Realism to Aircraft Cockpit Simulator Displays

Kim Stults
*580th Software Maintenance Squadron*

*Testing operational flight programs (OFPs) for aircraft requires that the user be able to enter data and generate output via a graphical user interface (GUI). Desiring to enhance the realism for the test team and upgrade the outdated software, we began the search for a new tool. A new commercial off-the-shelf (COTS) product was selected and a success story unfolded. This article presents that story.*

In 1995, after a decade of development, the Special Operation Forces (SOF) Extendable Integration Support Environment (EISE) was established at Robins Air Force Base in Warner Robins, Georgia. The purpose of the SOF EISE was to provide hardware and software support for seven selected avionics systems (see Figure 1). The support environment permits the modification and test of the OFPs running on the aircraft.

A key component of the SOF EISE is the crew interface (CIF) simulation. It is designed to allow real-time, simultaneous operations with the line replaceable unit and environment simulation portions of SOF EISE. The computer display touchscreens provide a simulation of the actual aircraft cockpits. The CIF stands as the key control element between the system user and both the real and simulated aircraft avionics. It provides a means by which the functional capability of simulated aircraft avionics, real aircraft avionics, and their associated OFPs can be evaluated against the required capability for these system components.

## Problem

After almost 20 years, the architecture of the CIF segment was becoming unsupportable. A unique CIF simulation executed on a single-board computer (SBC) for each of the seven systems. The SBC code communicated with two silicon graphic computers over a network using a remote procedure call protocols (RPC). Three processes ran on one of the silicon graphics (client, server, and GUI), and two processes ran on the second (client and GUI). These processes communicated using COTS software library calls. The hardware was becoming unsupportable and the software solutions were outdated. The segment was incredibly complex and resulted in reliability problems as well as high maintenance costs.

Figure 1: *SOF EISE Supported Aircraft*

| SOF Aircraft | Deployment Date |
|---|---|
| MH-53J PAVE LOW III (PL3) | 1995 |
| AC-130H Gunship (GS) | 1997 |
| MC-130H Combat Talon II (CT2) | 1998 |
| MC-130E Combat Talon I (CT1) | 1999 |
| MH-53M Pave LOW IV (PL4) | 1999 |
| EC-130H Compass Call (CC) | 2003 |
| HH-60G PAVE HAWK (PH) | 2006 |

## Potential Challenges

Even though a COTS product had been in place, high maintenance cost, poor customer support, and an unreliable development tool led to an investigation of new possibilities for today's technology for both hardware and software. Our requirements fell into basically two categories:

1. **Functional requirements.** These consisted primarily of requirements elicitation from our test group customer. The test group had written extensive procedures over the years that referred to specific graphical objects on the interface. The steps included *observe* actions that specified color, position, button presses, and other very specific details of the legacy graphics. This was a justifiable constraint given the level of effort that would be required to change thousands of pages of test procedures.
2. **Non-functional requirements.** These are essential to retarget the CIF GUI that resided on a silicon graphics machine to a Linux based system, better performance, and higher reliability.

### *COTS Considerations*

After it was decided that the functional capabilities of the legacy system would determine the requirements for the upgrade, a development team met to discuss the vision for the upgrade. Discussion focused on previous problems with vendors, ease of debugging, and the future of the system. From this discussion, it was determined that we were looking for the following [1]:

- **A fully interactive 2-D or 3-D open graphic library-based development tool.** We needed a tool that would allow us to create custom widgets that looked exactly like the legacy widgets. Only tools with the ability to create 2-D objects could do that. We also wanted the capability to improve the appearance of those objects that did not impact test procedures. In the future, we will need to support new platforms. For those new platforms, we will not be restricted to legacy appearances. For these, we would like to create more photo-realistic panels. There is a growing desire for out-the-window views by the users. Should that ever become a requirement, we want to be able to meet it without having to change tools. A 3-D tool is required for that.
- **Non-proprietary, human-readable, object-oriented code.** Historically, there have been issues with the legacy tool that required us to examine the interim files that were generated. Because the interim files were written in a proprietary format, we were unable to analyze certain conditions while debugging. This significantly hampered our ability to quickly correct some defects, and we were not anxious to subject ourselves to that limitation again.
- **Lower development cost.** The legacy tool was old and its customer base was decreasing. Licensing fees were increasing. Expertise with the tool was limited. All these factors forced the cost of the development tool to rise. Newer, cheaper, and more capable tools existed. We would take a cursory look at a few of them and determine which of those deserve further evaluation.
- **Efficient object-oriented designs and code generation.** An object oriented approach to programming is accepted industry-wide. While it is not a function of the tool to provide the

approach, some tools make it easier than others. We wanted a tool that would support, even enforce, object oriented code.

- **A compact runtime library.** Using our legacy tool, the libraries had become nearly unmanageable due to increasing size. While the increases were mandated by increased capability, they were still consuming disk space.
- **Flexible licensing options.** As the number of supported platforms grows, we needed to be able to adjust our licensing agreements. We needed to be able to establish a fixed number of development licenses and a different number of runtime licenses. We needed the option of a site license, in the event the requirements dictated. That is, in the event the number of platforms we were required to support grew to the point that it was economically feasible to get a site license instead of individual run-time license.
- **Proven COTS product with some demonstrated level of maturity.** There are a plethora of tools on the market that meet our needs. Many of them are excellent, some are only good. We had neither the time, nor the engineering resources to evaluate all, or even most of them in depth. We elected to rely on the tool's customer base to do that for us. Mature tools have a large customer base. We wanted an alternative pool for advice, customers who were already using the tool.
- **COTS vendor with good technical support.** Every new tool comes with the promise of technical support. We wanted to be sure that we were getting more than promises. The vendor we eventually selected provided excellent technical support during the evaluation period. We described several unique problems and they promptly provided explanations or solutions.

Certainly there are a lot of graphics products on the market. We looked at several and had vendors come in to demonstrate their products. We obtained references from those companies and contacted their customers. One can usually learn more from a vendor's customer than a vendor representative. For example, you can find out what to expect in the company's training courses, the quality of the technical support, the tool's ease of use, etc. After making an initial company selection for prototyping, we arranged for a week-long training session at a facility near our site. The instructor was technically competent and a veteran in front of the classroom. He provided satisfactory answers and examples for all of our questions. Most of the questions were drawn from what we thought would be difficulties with using his company's tool. In every case, we were more than satisfied with his response.

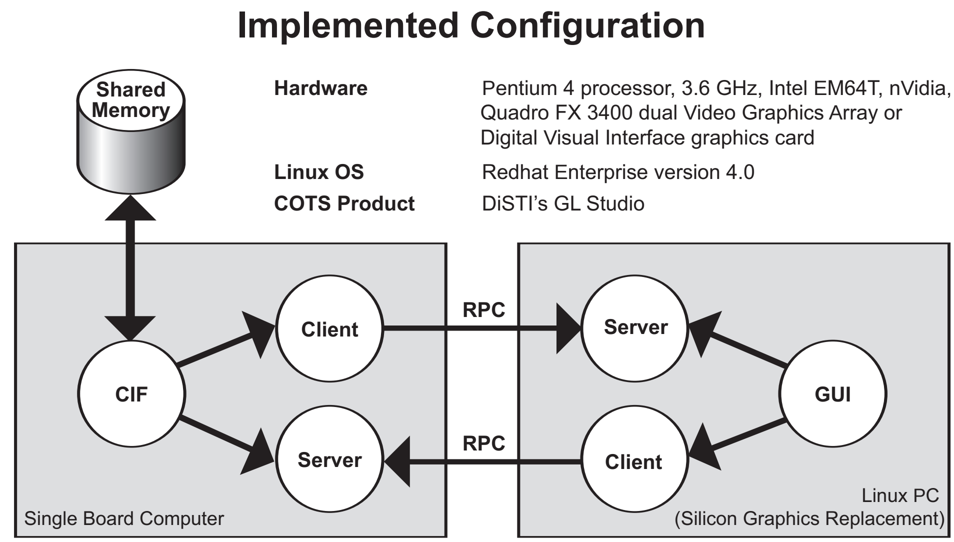



Figure 2: *Selected Architecture for Configuration*

## Obstacles

Change is not always welcome. There were concerns in moving from the familiar to the unfamiliar. We had previously faced difficulties with customer support, so a good working relationship with the vendor was crucial. There were concerns from the test team that the look and feel of the test station not be changed for fear of impact to the current test procedures.

The other obstacles we encountered were typically hardware-related or self imposed. The original Linux personal computer we were given for development had an incompatible operating system (OS) and graphics card for the tool we were evaluating. Our first step was to acquire a compatible OS. At this point though, the hardware acquisition still lagged behind the software we were developing. The integrated product would be driving two monitors. We did our early development on systems that only had one. We were concerned that mouse clicks on the second monitor would not return the screen coordinates we were expecting. We tasked one of our developers to examine the possibility of intercepting the universal serial bus event stream and modifying it. This was a risk mitigation step. Our next step was to choose a multi-headed graphics card to work with our selected OS. Finally, we had to find a touch screen that had existing drivers for our chosen Linux OS. Once all of our hardware was in place, we knew that the (x,y) desired coordinates were being returned from mouse events and the dual screen implementation was a non-issue. The vendor had assured us that this would not be a problem, but we had no way to prove it. Therefore, a lot of time had been spent on risk mitigation that was completely unnecessary.

## The Solution

The ultimate architecture for our solution was a combination of procedural programming in the system simulation and object-oriented programming in the GUI. Figure 2 depicts the selected architecture for our solution. DiSTI GL Studio was a perfect fit for our COTS product requirements. GL Studio is a premier Human Machine Interface (HMI) development toolkit that allows for the creation of end-to-end safety critical displays from prototype to delivery. GL Studio has no proprietary formats and flexible licensing options, and it produces reliable, safe, efficient, reusable applications in a rapid and easy fashion. The GL Studio development package was a fraction of the cost of our previous COTS product. The licensing fees alone will save approximately $120,000 over the next five years. The PCs will save an additional $42,000 over the same time period.

## Keys to Successful Integration

Our COTS integration was completed ahead of schedule and within budget. The integration effort was a success story for many reasons.

### *Partnering With the Vendor*

We received and continue to receive excellent technical support from DiSTI. Some vendors have a take-it-or-leave-it approach. We were a team and they wanted us to succeed. We were not the only

ones working weekends when we had a problem to solve. Particularly, they worked one weekend to develop a drop-in keypad class for us. In the beginning, we had issues with the particular OS we had chosen. It was not supported by GL Studio, but they helped us get it working anyway. They also had many customers running multi-headed touch screen applications in Windows, but we were the first to attempt it with Linux. Thanks to DiSTI's support, we were able to implement our desired design.

### *Networking With Similar Users*

DiSTI networked us together with the C-130 Self-Contained Navigation System (SCNS). ARINC had previously designed photo-realistic pilot panels for the SCNS project. We were able to obtain the reusable software objects for several pilot instruments and implement them into our architecture seamlessly. This saved a tremendous amount of time and money. For our initial delivery, we did not have time in the schedule to photograph our own instruments and code the displays. Having a product that was already being used with displays that we needed was a life saver.

### *Selecting the Hardware*

Implementing the design of one multi-head PC with two monitors – on each of the five platforms – reduced the 10 required host computers to five. This single PC architecture, along with utilizing a shared memory segment to replace the outdated communications library, greatly simplified our code – eliminating approximately 25,000 lines of code.

## Assessment of Results

The overall results for our upgrade have been outstanding. At this time, all of the platforms have been successfully ported to the new architecture and have been in use for approximately six months. Not only has the integration been a success for the development team, but for the end users as well.

### *Customer Satisfaction*

Even though our customer, OFP developers, and testers were leery of a change in the beginning, they have been very pleased with the improvements in our architecture, the most notable improvement being the photo-realistic displays for the pilot instruments (Figure 3, a and b). This, of course, enhances the realism of flight simulation and test.

> *"Use of COTS products is a viable way to upgrade existing systems."*

### *Sharing Across Multiple Platforms*

A huge success of the EISE in general is the ability to share software across multiple platforms. The object-oriented design approach saved us a substantial amount of time by providing the capability to reuse components instead of redeveloping individual instruments or instruments parts. The new COTS software also supports multiple OSs, leaving the future possibility for further upgrades and configuration changes completely open and easy to maintain.

## Conclusion

Use of COTS products is a viable way to upgrade existing systems. As part of our development plan, the port was completed in incremental steps. In order to minimize risk, our first step was to port only the code that ran on the silicon graphics machine and to maintain the original interfaces between the SBC code and the silicon graphics. Once the new graphics were in place, the second phase was to restructure the SBC code and make minimal interface changes. This approach worked very well for our initial delivery.

Now that we have transitioned to the new architecture, we have many options available to pursue. The most important option is the addition of more photo-realistic displays as time and schedule permits. The Linux solution was a good fit with the planned migration to a real-time Linux architecture in the lab. In general, the COTS HMI development tool was able to help us save a substantial amount of time and budget, allowing us to complete our objective well within our required deadline. The new architecture of the instrumentation lends itself for use in many other applications for the future as well, further saving time and budget for the SOF EISE lab as well as any other entity that may receive this source as government-furnished information for other similar programs.◆

## References

1. "Generating Human Machine Interfaces." Orlando: DiSTI, 2006.

Figure 3 a and b: *CT1 Pilot Display*


## About the Author

**Kim Stults** is a member of the technical team for the 580th Software Maintenance Squadron (580 SMXS) of the 402d Software Maintenance Group at Warner Robins Air Logistics Center, Robins Air Force Base, GA. The 580 SMXS performs software maintenance changes for the SOF weapon systems. She is responsible for the crew interface segment of the EISE. Stults has 17 years experience in software engineering, 11 in private industry, and six with the Air Force. She has bachelor's degrees in mathematics and computer science.

**420 Richard Ray BLVD**
**STE 100**
**Robins AFB, GA 31098**
**Phone: (478) 926-0718**
**Fax: (478) 926-0226**
**E-mail: kimberly.stults@robins.af.mil**

# Applying COTS Java Benefits to Mission-Critical Real-Time Software

Dr. Kelvin Nilsen
*Aonix*

*As mission-critical, real-time software systems grow in size and complexity, there is increasing pressure to incorporate commercial off-the-shelf (COTS) components as part of the strategy for reducing the total costs of developing and maintaining these systems. In the traditional information technology (IT) space, the object-oriented features of the Java language have proven their value in enabling significant increases in software reuse, and Java has now overtaken C and C++ as the language of choice for most enterprise IT projects. This article discusses the use of Java in mission-critical, real-time systems and emphasizes approaches that address common requirements for portable, efficient, responsive, and predictable real-time systems. These approaches make it possible to develop both soft and hard real-time components, which become COTS real-time components for integration within future real-time systems. Also supported is the ability to integrate non-real-time COTS Java components within systems that have high reliability and real-time constraints.*

Moore's law [1], proven by more than four decades of experience, describes the well-known phenomenon that, for a given price point, commercially available computational capacity doubles every 18 months. This increasing computational capacity is used by application software to provide improved functional capabilities, to respond to increasing numbers of service requests with shorter response times, and to support more efficient operations (better fuel efficiency, reduced system failures and down time, and increased productivity). Studies of embedded system trends demonstrate that the size of software in embedded systems also grows exponentially, doubling in size every 18-36 months, depending on the industry [2, 3].

Many of today's typical embedded real-time systems are comprised of hundreds of thousands, if not millions of lines of code. Rather than develop all of the code required for each new product release from scratch, the majority of today's software growth results from integration of third-party software components and the melding of independently developed software systems into larger integrated systems offering the combined capabilities of each individual part.

This article discusses the use of the real-time Java language as an enabling technology to greatly reduce the efforts associated with developing and maintaining reusable real-time software components. The discussion also addresses the need to integrate within real-time systems certain components that were not originally developed with the rigor or discipline typical of real-time software. These non-real-time software components can be successfully deployed within real-time systems as long as systems are carefully constructed to assure that less disciplined components do not steal resources from the allotments for real-time components or compromise the timely execution of them.

## What Is Real-Time Software?

The correctness of a real-time system depends not only on delivering correct computational results, but also on delivering these results at the correct time. If results are delivered too early or too late, the real-time system is operating incorrectly. It is the software developer's responsibility to assure that the system operates correctly. Real-time software can be divided into two broad categories: *hard real-time* and *soft real-time*.

> *"The resource needs of each hard real-time component are determined through careful theoretical analysis ... ."*

*Hard real-time* systems are developed according to the most stringent and conservative practices [4]. The resource needs of each hard real-time component are determined through careful theoretical analysis of the worst-case central processing unit (CPU) time and memory consumption along every worst-case path though the code. On modern CPU architectures, this results in very conservative use of computing resources.

*Soft real-time* systems comply with real-time constraints using empirical rather than analytical techniques [4]. Characterizing each component's resource needs statistically, developers use probability theory to assess the likelihood that a system integrating multiple independently analyzed components will meet all of its real-time constraints. Since soft real-time systems are not proven with 100 percent certainty to always satisfy all real-time constraints, soft real-time developers must design and implement contingency mechanisms to deal with the occasional missed deadline.

The term *safety-critical* describes software systems that must be certified to the satisfaction of government regulatory auditors because human lives may be lost if the software malfunctions [5]. Such software plays critical roles in commercial avionics, passenger rail systems, nuclear power plants, and certain medical equipment. The rigor of safety certification calls for extremely conservative development practices. As described in this article, safety-critical Java is a proper subset of hard real-time Java technologies.

## Using the Java Programming Language for Real-Time Development

The appeal of Java derives from improved developer productivity, reduced software maintenance costs, higher software reliability, enhanced functionality, and improved generality, all of which lead to expanded software longevity. In the traditional business information processing marketplace (financial record keeping, customer relations management, inventory controls, billing, payroll, etc.), the Java programming language has replaced C++ as the predominant programming language, largely because Java programmers are approximately twice as productive when developing new code and are five to 10

times as productive during maintenance of existing code [6-8]. Various real-time Java technologies extend these benefits into embedded real-time systems.

Much of the content presented in this article derives from the general recommendations available in *Guidelines for Scalable Java Development of Real-Time Systems* [9], a document originally developed to guide the use of real-time Java technologies by the European Space Agency. The document details three different approaches to the use of real-time Java, each tailored to the needs of a specific audience: soft real-time, hard real-time, and safety critical real-time. Safe and efficient mechanisms make it possible to build complex systems comprised of components implemented in each of the three different real-time Java profiles.

An important objective of this article is to help engineers understand the trade-offs in selecting between alternative technologies. Soft real-time Java technologies offer the greatest ease of development and maintenance and provide access to the largest existing availability of ready-to-use open-source and COTS Java components. The more constrained hard real-time and safety-critical Java technologies are more difficult for programmers to use, but they offer improved determinism and much more efficient deployment. They also represent a much simpler run-time environment, facilitating the creation of safety-certification artifacts.

## Developing Reusable Soft Real-Time Components

One of the key reasons why Java developers are more productive than C and C++ developers is because of automatic garbage collection. According to a study performed by Xerox Palo Alto Research Center in the early 1980s [10], automatic garbage collection reduces programming efforts associated with large, complex software systems by approximately 40 percent. These benefits are amplified significantly in the Java environment because automatic garbage collection is the foundation on which millions of lines of COTS software, including all of the standard Java libraries, are based. Removing garbage collection from the Java language makes it more difficult to develop new software and also precludes the use of nearly all existing Java library code.

However, the power of garbage collection comes with a cost. Traditional Java implementations occasionally pause execution of all Java threads to scan memory in search of objects no longer in use. These pauses can last tens of seconds with large memory heaps. Memory heaps ranging from 100 Mbytes to multiple gigabytes are currently being used in mission-critical systems. The 30-second garbage collection pause times experienced with traditional Java Virtual Machines (VMs) are incompatible with the real-time execution requirements of most mission-critical systems. Special real-time VMs support pre-emptible and incremental garbage collection. This approach is suitable for soft real-time systems with timing constraints as low as a few hundred microseconds.

One of the costs of automatic garbage collection is the overhead of implementing shared protocols between application threads. Application threads continually modify the way objects relate to each other within memory while garbage collection threads continually try to identify objects no longer reached from any threads in the system. This coordination overhead is one of the main reasons that compiled Java programs run at one third to one half the speed of optimized C code.

> *“When developers speak of hard real-time software, they generally expect that the software be proven to satisfy all real-time constraints prior to execution.”*

The complexity of the garbage collection process and of any software depending on garbage collection for reliable execution is beyond the reach of cost-effective static analysis to guarantee compliance with hard real-time constraints. Thus, real-time garbage collection is recommended for soft real-time but not hard real-time systems.

Portable soft real-time components should adhere to the following guidelines:

1. Use standard edition Java Application Programming Interface as this provides access to the most widely available set of built-in services.
2. Instrument the component so it can determine its memory and CPU-time requirements on a given deployment target.
3. Deploy the software on a real-time VM that supports fixed-priority scheduling, priority inheritance, priority-ordered queues, and real-time garbage collection.

Software constructed according to these conventions is easily retargeted and integrated into a variety of soft real-time applications. Developers can perform resource needs analysis automatically as part of the dynamic class loading process or manually as part of the software maintenance and integration effort.

Many soft real-time systems have already been fielded using the approaches described in this section. Projects that publicly acknowledge their adoption of real-time Java approaches include Aegis Software System Upgrade [11], Boeing’s J-UCAS effort [12], Calix C7 Multi-Service Access System [13], the FELIN (Fantassin à Équipements et Liaisons Intégrés) wearable computer system [14], Nortel’s high-end, long-haul fiber-optic switch [15], and Varco’s robotic oil exploration system [16]. Developers consistently find these approaches to development and maintenance of soft real-time software result in significant developer productivity increases and software maintenance cost savings in comparison with the use of C or C++.

## Developing Reusable Hard Real-Time Components

When developers speak of hard real-time software, they generally expect that the software be proven to satisfy all real-time constraints prior to execution. Programmers who write hard real-time software expect their programming environment to support at minimum the following:

1. Standard libraries precisely constrained by worst-case CPU-time consumption and memory usage.
2. Programming language syntactic features for which the worst-case CPU-time and memory usage of the equivalent machine-language translations are precisely constrained.
3. Programming language and/or library services that allow developers to speak in very precise terms regarding the timing constraints imposed on particular real-time software components.
4. Development tools to assist with the analysis of worst-case CPU time and memory requirements for particular

software components.

Additionally, many developers of hard real-time code face additional design constraints beyond the need to prove compliance with real-time constraints. For example, developers face severe CPU time, memory footprint, and battery power constraints; safety certification requirements; marketplace competition that demands challenging functional upgrades; interoperability and maintainability requirements; aggressive completion schedules; and limited development budgets.

### *Enforcing Static Properties*

Java compilers perform more static property analysis to enforce stronger consistency checking within software systems than C and C++ compilers. This stronger consistency checking reduces programming errors, further improving developer productivity. Examples of built-in static property enforcement in Java include the following:

1. Strong type checking prohibits incompatible type coercion between, for example, an integer and a reference type, or between two incompatible reference types.
2. Programmers are prohibited from adding integer values to reference values to create references to new objects.
3. Byte-code verification assures that the types of actual parameters passed to a method invocation exactly match the types of the formal parameters declared for the invoked method.
4. When programmers write code that will throw an exception under certain conditions, the Java compiler enforces that every context that invokes this code is prepared to catch the exception. This reduces the likelihood that error conditions will be overlooked or ignored.

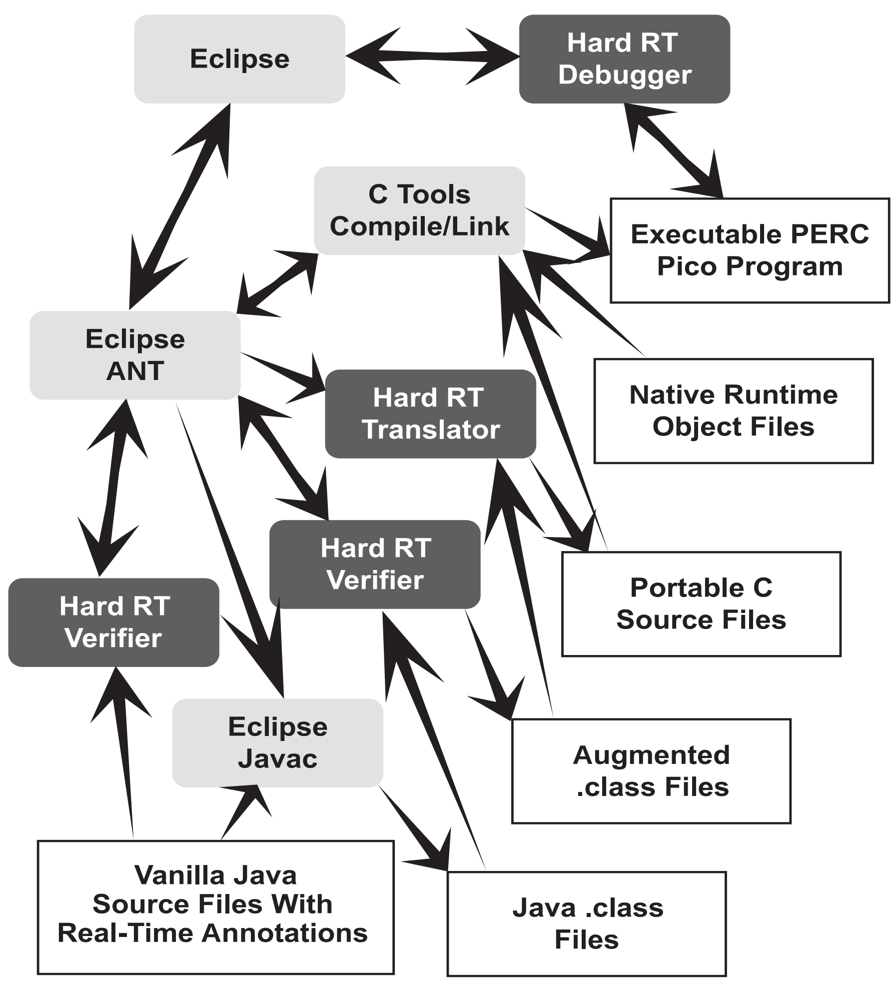
This consistency checking is especially helpful when large software systems are assembled from components independently produced by different teams of developers.

Because of the strong static property checking already enforced by the Java programming environment, Java has attracted the attention of software engineers as a possible language of choice for implementation of hard real-time and safety-critical systems. Hard real-time software includes radar and sonar systems, global positioning systems, software-defined radios, and low-level device drivers for various peripheral devices. Safety-critical software includes anti-lock braking systems in consumer vehicles, fly-by-wire control of flight surfaces in commercial aircraft, automatic shutdown of nuclear power plants, computer-controlled switching systems in passenger railroad systems, and weapons fire-control software.

To address the needs of hard real-time developers, the Open Group is sponsoring a Java community process expert group to establish standards for safety-critical development with the Java language. The Open Group is a vendor and technology-neutral consortium that works to enable access to integrated information within and between enterprises based on open standards and global interoperability. The intention is to produce a standard that is endorsed both by the Java Community Process and by the International Organization for Standardization (ISO). This standard is based on the existing Real-Time Specification for Java (RTSJ) [17]. As a key contributor to this standardization effort, Aonix has drafted a set of guidelines for real-time developers who desire to use the Java language [9]. Programmers who develop their code according to these draft guidelines for development of hard real-time and safety-critical Java can rely on assurances from a special byte-code verifier.

1. The maximum amount of CPU time required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program.
2. The maximum amount of stack memory required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program.
3. Execution of a particular method (or of any overriding method) will not allocate any memory in the shared immortal heap.
4. No blocking operations will be attempted while a thread holds a queue-free priority ceiling emulation lock.
5. Execution of particular methods will not result in throwing of RTSJ-defined CeilingViolationException, Duplicate FilterException, IllegalAssignment Error, InaccessibleAreaException,

Figure 1: *Hard Real-Time Java Development Environment*



MemoryAccessError, MemoryScope Exception, MemoryTypeConflictException, OutOfMemoryError, Scoped CycleException, StackOverflowError, or ThrowBoundaryError exceptions.

Enforcement of these static properties is provided by a special byte-code verifier that enforces more stringent constraints than the traditional Java byte-code verifier.

A proposed integrated development environment is illustrated in Figure 1 (see page 21). This hard real-time development environment builds upon the popular open-source Eclipse integrated development environment. COTS technologies, color coded in light grey, provide the foundation of the hard real-time development environment. Hard real-time plug-ins to the open Eclipse architecture, color coded as dark grey, provide special hard real-time development tool capabilities. The *Hard RT (Real-Time) builder* takes responsibility for determining which parts of a large software system need to be retranslated by the *Eclipse Javac* program and reverified by the *Hard RT verifier* each time a programmer modifies existing source code. Errors detected by either *Eclipse Javac* or by the *Hard RT Verifier* are highlighted within the Eclipse Java-syntax-directed editing window, providing immediate developer feedback, and simplifying the development process.

The functional behavior of hard real-time Java code can be exercised and debugged using a traditional Java 5.0 run-time environment. To test the hard real-time Java code on the target hardware, the verified Java class file is translated by the *Hard RT translator* to C code. Then it is compiled and linked with run-time libraries using COTS C-language development tools. C compilers are available to support all popular embedded processor architectures and real-time operating systems (OSs). The C code generated by the *Hard RT translator* can be loaded directly into read-only memory and can execute in place. The *Hard RT debugger* allows developers to debug the executable hard real-time program using a familiar Eclipse Java debugging environment, even though the deployed code was produced by C-language development tools.

Note that the hard real-time Java development environment translates Java code to C before deployment. This provides much higher performance (up to 3.5 times faster than comparable Java code running with the Sun Microsystems HotSpot compiler), much smaller memory footprint (more than 10 times smaller), and tighter real-time latencies (microseconds vs. tens of seconds) than traditional Java. Comparisons with the performance of handwritten C code reveal that this technology generally produces code that runs within 35 percent (either faster or slower) of comparable handwritten C code. The special hard real-time Java verifier automates the static analysis that must be performed by non-standard, third party tools when using less structured languages like C and C++. Because the hard real-time Java verifier is tightly integrated with other components of the development tool chain, the development and deployment process is much smoother. Information flows automatically between the syntax-directed editor, the Java compiler, the hard real-time byte-code verifier, and the C compiler that produces the final machine-code translation of the original real-time Java program.

*“In summary, the hard real-time version of Java allocates all objects on the runtime stack of the current thread rather than using a garbage-collected heap.”*

When hard real-time Java technologies are used to implement safety-critical systems, the hard real-time Java verifier imposes additional constraints beyond the constraints that are imposed on typical hard real-time software. These additional constraints, motivated by established safety certification guidelines, help enforce that all programmers who contribute to the development of a safety-critical software system adhere to the same conservative development practices.

The most popular alternative approach to development of hard real-time code involves the use of Misra C [18]. In comparison to Misra C, the hard real-time Java approach offers superior portability, scalability, and maintainability. It also provides much easier, more reliable, and more efficient integration with higher level Java software which is typical of the much larger non-real-time and soft real-time layers of many mission-critical software hierarchies. Another key advantage of developing and maintaining hard real-time code with Java tools is that today’s graduating software engineering students are much more likely to be proficient in Java than in any other programming language. When compared with Misra C, the key disadvantages of the hard real-time Java approach are that the technology is newer and less familiar to established safety-critical domain experts, and the hard real-time Java approach introduces an additional abstraction layer between source code and deployment. This additional abstraction layer contributes to ease of software maintainability, portability, and software scalability, analogous to abstraction improvements provided by C over assembly language. Some additional effort may be required to trace requirements through source code and multiple intermediate code representations to the final machine code implementation.

### Memory for Temporary Objects

Since Java is an object-oriented programming language, all structured data is represented by objects. With a traditional Java run-time environment, all objects are allocated within a region of memory known as the heap, and the memory for these objects is reclaimed by an automatic garbage collector. For hard real-time development, the run-time environment does not include an automatic garbage collector.

The hard real-time Java guidelines allow Java components to allocate objects on the run-time stack using programming constructs similar to those of the C and C++ programming languages. Unlike C and C++, statically enforced programmer annotations guarantee that the use of stack-allocated memory does not create dangling pointers.

In summary, the hard real-time version of Java allocates all objects on the run-time stack of the current thread rather than using a garbage-collected heap. Any running thread may spawn additional threads by dedicating a portion of its stack to represent the run-time stack of the spawned thread. Any objects that need to be shared between multiple threads must be allocated within the stack of some ancestor thread.

The key benefits of safely allocating temporary objects on the run-time stack are the following:

- Temporary memory allocation and

deallocation is very fast.

- Temporary memory allocation is very reliable, because the stack never becomes fragmented, the maximum stack size can be determined through static analysis, and the absence of dangling pointers is guaranteed through the use of enforceable programmer annotations.

For hard real-time software, this gives Java developers capabilities and performance comparable to more traditional languages like Ada, C, and C++.

## Synergy Between Java Technologies

The embedded real-time market has been described as a thousand different niches. Each critical software component represents different requirements and economic trade-offs. Figure 2 provides a decision tree to assist system architects in deciding how to implement particular capabilities.

Notably, soft real-time Java development guidelines are generally preferred unless there is a specific constraint that precludes them because soft real-time Java offers the highest developer and software maintenance productivity.

The hard real-time Java technologies do not use automatic garbage collection. Instead, dynamic memory is allocated and deallocated under more explicit programmer control. The safety-critical Java standard supports a subset of the hard real-time Java capabilities.

Note that this decision tree does not distinguish between different security requirements. Security issues are largely orthogonal to real-time issues and are not the focus of this article. Techniques for enforcing multiple independent levels of security are compatible with hard, soft, and safety-critical Java technologies.

### *Selective Sharing of Control With Traditional Java Components*

The recommended approach for providing efficient and reliable integration of hard real-time components with traditional Java components uses a restrictive form of object sharing between the hard real-time and traditional Java domains. The shared objects always reside in the hard real-time domain and do not participate in garbage collection. Since these are hard real-time objects, they are never subject to relocation. This greatly simplifies the implementation and improves execution efficiency.

We describe object sharing as *restrictive* because the traditional Java domain cannot see any of the instance or static variables associated with the hard real-time object. Furthermore, it cannot see the object's regular methods. It can only see methods specially designated as traditional Java methods. These *traditional Java methods* are analogous to OS entry points for application software. In this regard, writing hard real-time Java code is similar to making modifications to an operating system kernel. As with traditional operating system design, greater trust is placed in the implementers of the lower-level OS software, and great care is taken to ensure that errors or malicious intent of application software do not compromise the integrity of lower level components.

In traditional OS design, invocation of kernel services generally crosses a memory protection barrier, and hardware memory management units assure that application code cannot see or modify kernel code and data structures. The restrictive object sharing architecture supports the same abstraction guarantees, but it does so using static byte-code verification techniques which allow much more efficient integration of the hard real-time and non-real-time software. The performance benefits of this architecture have already been demonstrated, for example, in studies conducted by Calton Pu on the Synthesis Kernel [19].

When systems are comprised of a combination of hard and soft real-time components, the hard real-time components will typically run with footprint and throughput efficiency very close to that of optimized C. This represents a three-fold improvement over typical optimized Java performance and footprint. There are many important mission-critical needs that can be addressed by this configuration, such as the following:

- Portable and very efficient device drivers (possibly, but not necessarily, having hard real-time constraints).
- Compared with the use of the Java Native Interface (JNI), interfaces to legacy (*native*) components written in other languages are much more efficient and much safer if implemented using the hard real-time Java technologies as an intermediary between traditional Java and the native code.
- Performance-critical code such as Fourier analysis and matrix manipulation can be provided much more efficiently as hard real-time Java components than as traditional Java code or as legacy code interfaced to Java through JNI.

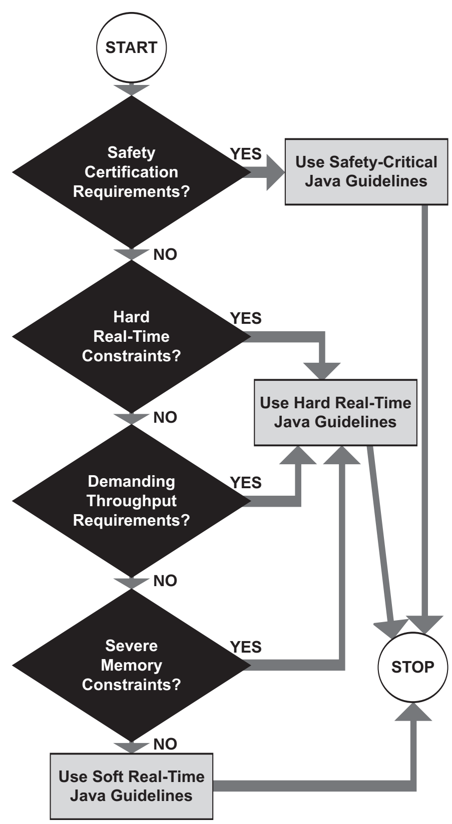
The hard real-time Java approaches described here are only now becoming commercially available, so current experience is limited to research experiments. A number of commercial and defense projects are beginning development based on these technologies. Information should be available in another year or so.

## Using Off-the-Shelf Components in Real-Time Systems

The key to using traditional off-the-shelf Java components in real-time systems involves careful partitioning of capabilities to ensure that reliable operation of real-time components is not compromised by the less-disciplined behavior of non-real-time components. System architects and integrators need to evaluate which partitioning approaches are most appropriate for each particular application's requirements. Among the practices in common use today are the following:

1. Structure the application so non-real-time code runs in a different VM (possibly even on a different processor) than real-time code. By isolating the non-real-time code within a distinct VM, it is possible to enforce strict memory budgets and limit the sched-

Figure 2: *Decision Tree for Selecting Between Alternative Java Technologies*



uling priorities at which the non-real-time components consume CPU time.

2. Allow selected non-real-time components to run on the same VM as more carefully constructed soft real-time components only after thorough testing of the non-real-time software to establish sufficient confidence that the resource needs of this non-real-time software are well understood.
3. Take extra care to assure that real-time software does not have to wait indefinitely for interactions with non-real-time software. For example:
   a. Run the non-real-time software within a different VM than the real-time software.
   b. Avoid blocking on synchronization for objects that are shared between non-real-time and real-time components.
   c. OR make use of alternative information sources (e.g. approximations) whenever a non-real-time component does not deliver critical information to the real-time component within the window of time that is required in order to satisfy end-to-end timing constraints.
4. Run components with the most stringent real-time constraints within a hard real-time environment at priorities higher than all soft real-time components. The hard real-time environment enforces strict memory partitioning to prevent memory contention from non-real-time components running in the traditional Java VM environment.

These are the approaches that have been successfully deployed in the various projects mentioned in [11] through [16].

## Conclusions

By restricting the use of Java programming language features and libraries, and by exploiting special static analysis tools, it is possible to apply many of the benefits of the Java programming language to the specialized domain of real-time software. Standards to support these development approaches are being developed under the auspices of the open group, which is working to establish standards that can be endorsed both by the Java Community Process and by ISO.

By carefully partitioning functionality between high and low-level software, it is possible to leverage the best strengths of Java within each respective programming domain. Lower-level Java technologies are most appropriate for implementing low-level device drivers, interrupt handlers, safe and efficient interfaces to legacy (*native*) code, and certain performance-critical components such as Fast Fourier transforms. Higher-level Java technologies are most appropriate for larger, more complex functionality, especially subsystems that need to support dynamic reconfiguration and/or generic reuse of existing off-the-shelf Java software components.◆

## References


1. Moore, G. "Cramming More Components Onto Integrated Circuits." Electronics Magazine Apr. 1965.
2. Bourgonjon, R. "The Evolution of Embedded Software in Consumer Products." International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, 1995.
3. McMillan, R. "GM CTO Sees More Code on Future Cars." InfoWorld Oct. 2004.
4. Jensen, E.D. "Overview of Fundamental Real-Time Concepts and Terms." Real-Time 19 Feb. 2007 <www.real-time.org/realtimeoverview.htm>.
5. "Software Considerations in Airborne Systems and Equipment Certification." RTCA/DO-178B. RTCA, Inc., 1992.
6. Dios, Chen, et al. "An Empirical Study of Programming Language Trends." IEEE Software 22.3 (2005).
7. Nilsen, K. "Quantitative Analysis of Developer Productivity in C vs. Real-Time Java." Defense Advanced Research Projects Agency Workshop on Real-Time Java, 13 July 2004, Arlington, TX: Aonix Research and Development, 2004.
8. Phipps, G. "Comparing Observed Bug and Productivity Rates for Java and C++." Software Practice and Experience 29.4 (1999): 345-358.
9. Nilsen, K. "Guidelines for Scalable Java Development of Real-Time Systems." Aonix, 2006. <http://research.aonix.com/jsc/rtjava.guidelines.3-28-06.pdf>.
10. Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly Typed, Statically Checked, Concurrent Language." CSL-84-7. Xerox Palo Alto Research Center, 1985 <www.parc.xerox.com/about/history/pub-historical.html>.
11. "Lockheed Martin Selects Aonix PERC Virtual Machine for Aegis Weapon System." Space War 13 Oct. 2006.
12. "Boeing Selects Software for J-UCAS X-45C." Defense Industry Daily 31 Oct 2005.
13. "Case Study: Centralized Network Element Platform, Calix Networks." Aonix <www.aonix.com/pdf/PERC_CalixSuccess.pdf>.
14. McHale, J. "Wearable Computers and the Military: The Smaller the Better." Military and Aerospace Electronics Nov. 2005.
15. "Case Study: Optical Network Switch, Nortel." Aonix <www.aonix.com/pdf/PERC_NortelSucces.pdf>.
16. "National Oilwell Varco Selects Aonix PERC for Java-Based Robotic Drilling." Rigzone 25 Sept. 2006 <www.rigzone.com>.
17. Bollella, G., et al. The Real-Time Specification for Java. Addison-Wesley, 2000.
18. "MISRA-C: 2004 – Guidelines for the Use of the C Language in Critical Systems." MISRA, 2004.
19. Pu, C., H. Massalin, and J. Ioannidis. "The Synthesis Kernel." Computing Systems 1.1 (1988): 11-32.


## About the Author

**Kelvin Nilsen, Ph.D.,** is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and COTS safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. In 2003, Aonix acquired NewMonics. Nilsen has a bachelor's degree in physics from Brigham Young University and a master's degree and a doctorate in computer science from the University of Arizona.


**Aonix**
**5930 Cornerstone Court West**
**STE 250**
**San Diego, CA 92121**
**Phone: (801) 756-4821**
**Fax: (801) 756-4839**
**E-mail: kelvin@aonix.com**

# The Relative Cost of Interchanging, Adding, or Dropping Quality Practices

Bob McCann
*Lockheed Martin Aeronautics*

*In developing systems and software, there are multiple opportunities to perform quality practices to find and to fix defects prior to putting the system or software into operations. This article demonstrates the following conclusions: 1) In general, quality practices should be ordered by increasing average cost to find and fix defects. Fixed costs do not affect this conclusion, but significant differences in either defect detection effectiveness or in the effectiveness of verifying rework induced defects can modify the conclusion. 2) One should retain or add a second quality practice provided the second practice fixes more defects during rework than the second practice creates during rework, provided the average cost to fix defects downstream is much larger than both the second practice's fixed costs and the second practice's marginal cost to find and fix defects.*

In the beginning of a new project, project management gets to decide a very important issue, namely what work products are subject to a verification process, e.g., peer reviews, formal inspection, testing, etc. Some work products may even be deemed sufficiently critical that they are subjected to multiple verification processes during the development life cycle (e.g., requirements inspection, design inspection, code inspection, code desk check, compile and fix, informal peer reviews of various kinds, and various flavors of testing).

In these cases, there is nearly always a discussion of whether to use an informal peer review instead of a formal inspection and whether or not to skip the pre-compile desk check or to perform the code inspection before or after the first successful compile or even after the completion of unit testing. What is always present is the persistent nagging feeling that too much or too little was spent on verification. This article, together with the two previous articles on cost effectiveness of inspections[1], addresses that issue with a simple quantitative cost analysis model. It should be noted that this model is easily extended when a cause of variation in any of the factors become known.

In what follows, statistical reasoning is used (where that is not appropriate, the results may differ[2]). For instance, it is highly unlikely that a compilation test will discover a design defect (such as poor choice of algorithm) that results in a performance problem. In contrast, it is quite likely that a formal inspection of the design, a formal inspection of the code, and a formal performance test will all have a statistical likelihood of discovering that same design defect.

**Warning:** There may be simpler, more elegant proofs of the above results than what follows. If algebra or statistics give you a headache or other trauma, the author apologizes for any discomfort from what follows.

## Analysis

Suppose three or more adjacent quality practices that find and fix defects are performed in series (e.g., personal desk check, one-on-one peer review, compile and fix, formal inspection, unit testing, etc.[3]). Further suppose the cost effectiveness of each is measurable[4] (please note that cost can be any independent variable of interest: dollars, labor hours, schedule impact, etc.):

- Let $Q_j$ be quality practice *j* where *j* is 1, 2, …
- Let $F_j$ be the fixed and sunk costs of quality practice $Q_j$[5]. Presumably $F_j$ will be small compared to other cost terms if there are a significant number of defects.
- Let $C_j$ be the average cost per defect found and fixed for practice $Q_j$ including verification practice $V_j$.
- Let $E_{Qj}$ be the average effectiveness – fraction of defects present found for practice $Q_j$. Thus $E_{Qj}*C_j$ would be the probable cost of finding and fixing a defect with quality practice $Q_j$.
- Let $E_{Vj}$ be the average effectiveness – fraction of defects present found for practice $V_j$. Thus $E_{Vj}*C_j$ would be the probable cost of finding and fixing a defect with verification practice $V_j$.
- Let $I_{Rj}$ be the number of defects inserted by the rework due to $Q_j$.
- Let $I_0$ be the number of defects inserted by earlier development practices. There is no breakage in this analysis if we only consider defects that are discovered sometime in the product life cycle. Defects that never get exercised have no actual impact, just potential impact.
- Let $II_0$ be the number of defects entering the second quality practice.
- Let $III_{ij}$ be the number of defects escaping both when $Q_i$ precedes $Q_j$.
- Let $R_j$ be the rework activity associated with quality practice $Q_j$.
- Let $V_j$ be the average effectiveness of the verification of rework performed in $Q_j$. For algebraic simplicity we will assume $V_j$ is approximately equal to $E_j$.
- Let $T_{LC}$ be the total life cycle cost with subscripts indicating what quality practices are performed and in which order. Note that the letters *TLC* can also mean *Tender Loving Care* throughout the full life cycle, which is what this author thinks is necessary to comply with the intent of section 804 of the Bob Stump Act of 2003[6]. Thus, $T_{LC12}$ refers to the case where $Q_1$ is performed before $Q_2$.
- Let $T_{LC12}$ be the total life-cycle cost when both $Q_1$ and $Q_2$ are present.
- Let $T_{LC1\_}$ be the total life-cycle cost when $Q_1$ is present and $Q_2$ is absent.
- Let $T_{LC\_2}$ be the total life-cycle cost when only $Q_2$ is present and $Q_1$ is absent.

Figure 1 shows the defect flow associated with quality practice $Q_1$. In Figure 1 the circles represent tasks, and the boxes represent collections of defects.

- $Q_1$ is a quality practice that finds a fraction $E_{Q1}$ of the defects present in some set of work products.
- $R_1$ is the task that does the impact analysis and repair for each of the identified defects. This in principle may introduce a new set $I_{R1}$ of defects.
- $V_1$ is the verification task that follows the rework effort. It verifies the solutions to the defects discovered in $Q_1$ and finds a fraction $E_{V1}$ of the new defects.
- Corrected defects do not propagate further.
- Escaped defects propagate to downstream processes.

When we stack two quality practices in

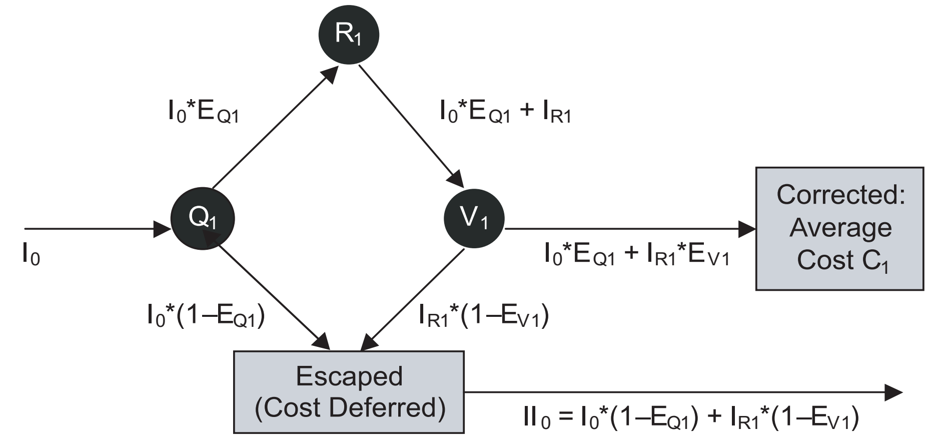



Figure 1: *Defect Flow for Quality Practice $Q_1$*

a row, the input to the second practice consists of the escaped defects from the first practice. Algebraically, this is accomplished by replicating Figure 1, changing the subscript 1 to 2 and replacing $I_0$ with $II_0 = I^0*(1-E_{Q1}) + I_{R1}*(1-E_{V1})$, as is shown in Figure 2.

Given the model described by Figure 1 and Figure 2, it is now algebraically possible to answer the following questions:

1. What is the cost increase in reversing the order of application of the two practices assuming all downstream defects eventually create an average cost $C_3$ per defect[7]?

$$\begin{aligned}
T_{LC\,12} &= F_1 + C_1*(I_{R1}*E_{V1} + I_0*E_{Q1}) + F_2 + C_2*(I_{R2}*E_{V2} + II_0*E_{Q2}) + F_3 + C_3*III_{12}\\
&= F_1 + C_1*(I_{R1}*E_{V1} + I_0*E_{Q1}) + F_2 + C_2*\{I_{R2}*E_{V2} + [I_0*(1-E_{Q1}) + I_{R1}*(1-E_{V1})]*E_{Q2}\} + F_3 + C_3*III_{12}\\
&= F_1 + F_2 + F_3 + C_1*I_{R1}*E_{V1} + C_2*I_{R2}*E_{V2} + C_1*I_0*E_{Q1} + C_2*I_0*(1-E_{Q1})*E_{Q2} + C_2*I_{R1}*(1-E_{V1})*E_{Q2} + C_3*III_{12}
\end{aligned}$$

$$\begin{aligned}
T_{LC\,21} &= F_1 + F_2 + F_3 + C_2*I_{R2}*E_{V2} + C_1*I_{R1}*E_{V1} + C_2*I_0*E_{Q2} + C_1*I_0*(1-E_{Q2})*E_{Q1} + C_1*I_{R2}*(1-E_{V2})*E_{Q1} + C_3*III_{21}
\end{aligned}$$

and

$$\begin{aligned}
III_{12} &= I_{R2}*(1-E_{V2}) + II_0*(1-E_{Q2})\\
&= I_{R2}*(1-E_{V2}) + [I_0*(1-E_{Q1}) + I_{R1}*(1-E_{V1})]*(1-E_{Q2})\\
III_{21} &= I_{R1}*(1-E_{V1}) + [I_0*(1-E_{Q2}) + I_{R2}*(1-E_{V2})]*(1-E_{Q1})
\end{aligned}$$

so

$$(III_{21}-III_{12}) = I_{R1}*E_{Q2}*(1-E_{V1})-I_{R2}*E_{Q1}*(1-E_{V2})$$

thus

$$\begin{aligned}
(T_{LC\,21}-T_{LC\,12}) &= I_0*(C_2*E_{Q2}-C_1*E_{Q1}) + I_0*[C_1*(1-E_{Q2})*E_{Q1}-C_2*(1-E_{Q1})*E_{Q2}] + C_1*I_{R2}*(1-E_{V2})*E_{Q1}-C_2*I_{R1}*(1-E_{V1})*E_{Q2} + C_3*(III_{21}-III_{12})\\
&= I_0*(C_2-C_1)*E_{Q1}*E_{Q2} + C_1*I_{R2}*(1-E_{V2})*E_{Q1}-C_2*I_{R1}*(1-E_{V1})*E_{Q2} + C_3*(III_{21}-III_{12})
\end{aligned}$$

$$(T_{LC\,21}-T_{LC\,12}) = I_0*(C_2-C_1)*E_{Q1}*E_{Q2} + C_1*I_{R2}*(1-E_{V2})*E_{Q1}-C_2*I_{R1}*(1-E_{V1})*E_{Q2}+ C_3*[I_{R1}*E_{Q2}*(1-E_{V1})-I_{R2}*E_{Q1}*(1-E_{V2})]$$

Dividing this equation by $C_3*I_0*E_{Q1}*E_{Q2}$ and setting the result to zero demonstrates the clarity of dimensionless ratios:

$$(T_{LC\,21}-T_{LC\,12})\ /C_3*I_0*E_{Q1}*E_{Q2} = 0$$

or

$$(C_2-C_1)/C_3 + [(C_3-C_2)/C_3]*(I_{R1}/I_0)*(1-E_{V1})/E_{Q1}-[(C_3-C_1)/C_3]*(I_{R2}/I_0)*(1-E_{V2})/E_{Q2} = 0$$

**(Equation 1)**

The solution to this equation divides the parameter space into two regions – one in which the interchange is cost effective and one in which it is not. When the left-hand side is positive, it is more cost effective to perform $Q_1$ before $Q_2$.

Although the first term is the one intuition would quickly identify, please note that because $C_3$ can be much larger than either $C_1$ or $C_2$, the second and third terms may dominate the outcome, especially during the operations and maintenance phase. Note that the fixed cost contributions all cancel exactly.

In general, the quality practices should be ordered by increasing average cost to find and fix defects. Fixed costs do not affect this conclusion, but significant differences in either defect detection effectiveness or in the effectiveness of verifying rework induced defects can modify the conclusion.

2. What is the cost of adding or dropping a quality practice?

   A. Suppose we add or drop $Q_1$:

$$\begin{aligned}
T_{LC\,12} &= F_1 + F_2 + C_1*I_{R1}*E_{V1} + C_2*I_{R2}*E_{V2} + C_1*I_0*E_{Q1} + C_2*I_0*(1-E_{Q1})*E_{Q2} + C_2*I_{R1}*(1-E_{V1})*E_{Q2} + F_3 + C_3*III_{12}
\end{aligned}$$

where,

$$III_{12} = I_{R2}*(1-E_{V2}) + [I_0*(1-E_{Q1}) + I_{R1}*(1-E_{V1})]*(1-E_{Q2})$$

$$T_{LC\_2} = F_2 + C_2*(I_{R2}*E_{V2} + I_0*E_{Q2}) + F_3 + C_3*III_{\_2}$$

where,

$$III_{\_2} = I_0*(1-E_{Q2}) + I_{R2}*(1-E_{V2})$$

so

$$(III_{\_2}-III_{12}) = [I_0*E_{Q1}-I_{R1}*(1-E_{V1})]*(1-E_{Q2})$$

thus

$$(T_{LC\_2}-T_{LC\,12}) = (I_0*E_{Q1} + I_{R1}*E_{V1})*(C_2*E_{Q2}-C_1) + C_3*[I_0*E_{Q1}-I_{R1}*(1-E_{V1})]*(1-E_{Q2}) -[F_1 + C_2*I_{R1}*E_{Q2}]$$

This case will require individual analysis using actual (or accurately estimated) cost performance data.

Keeping/adding $Q_1$ is better if

$$(I_0*E_{Q1}+I_{R1}*E_{V1})*(C_2*E_{Q2}-C_1) + C_3*[I_0*E_{Q1}-I_{R1}*(1-E_{V1})]*(1-E_{Q2}) > [F_1 + C_2*I_{R1}*E_{Q2}]$$

**(Equation 2)**

Indeed, if $C_3$ is sufficiently large and $I_{R1}$ is sufficiently small, it will always be practical to keep/add a quality practice. However, if $I_{R1}$ is sufficiently large, then the converse will be true. In this case, the cost incurred due to mistakes inserted during rework swamps the value of mistakes actually found and fixed. Under those conditions it is better to drop the (broken) quality practice.

It is also true that very high fixed costs can cause a quality practice to become impractical. This is especially true when the *cost* of primary concern is a very aggressive development schedule commitment. It takes serious discipline on the part of both development management and customer management to put long term goals before short term concerns. Recent congressional and Department of Defense efforts to emphasize total life-cycle costs appears to be an attempt to

provide a context in which this long term focus is even possible[7].

B. Suppose we add or drop $Q_2$:

$$T_{LC1\_} = F_1 + C_1*(I_{R1}*E_{V1} + I_0*E_{Q1}) + F_3 + C_3*III_{1\_}$$

$$T_{LC12} = F_1 + C_1*(I_{R1}*E_{V1} + I_0*E_{Q1}) + F_2 + C_2*(I_{R2}*E_{V2} + II_0*E_{Q2}) + F_3 + C_3*III_{12}$$

$$(T_{LC1\_}-T_{LC12}) = C_3*(III_{1\_}-III_{12})-F_2-C_2*(I_{R2}*E_{V2} + II_0*E_{Q2})$$

but $III_{1\_} = II_0$

so

$$(T_{LC1\_}-T_{LC12}) = C_3*\{II_0-[II_0*(1-E_{Q2}) + I_{R2}*(1-E_{V2})]\}-F_2-C_2*(I_{R2}*E_{V2} + II_0*E_{Q2})$$
$$= C_3*\{II_0*E_{Q2}-I_{R2}*(1-E_{V2})]\}-C_2*(I_{R2}*E_{V2} + II_0*E_{Q2})-F_2$$
$$= (C_3-C_2)*II_0*E_{Q2} + (C_3-C_2)*E_{V2}*I_{R2}-F_2-C_3*I_{R2}$$

We should retain/add $Q_2$ provided $(T_{LC1\_}-T_{LC12})>0$. This can be expressed as the following:

$$(C_3 - C_2)*II_0*E_{Q2} + (C_3 - C_2)*E_{V2}*I_{R2} > F_2 + C_3*I_{R2}$$

or

$$II_0*E_{Q2} + I_{R2}*E_{V2} > (C_3*I_{R2} + F_2)/(C_3 - C_2)$$

**(Equation 3)**

Therefore, one should retain/add $Q_2$ provided the second practice fixes more defects during rework than the second practice creates during rework provided $C_3$ is much larger than either $F_2$ or $C_2$.

## Worked Example

To make the results more solid, consider a software development effort delivering a million lines of code over five years by a team of 100 software developers. Given that software developers tend to change jobs quickly to keep their skills current, one can assume that defects found during design and coding will be fixed and verified by the original author and that defects found late in testing will be fixed and verified by someone other than the original author. Fixed and sunk costs will be ignored and some average error rates and costs for this team of developers will be guessed:

- Twenty defects per thousand lines of code inserted during coding and design: $I_0 = 20{,}000$.
- Inspection catches three out of every our defects present: $E_{Q1} = 0.75$.
- Three new defects are created for every 10 fixed: $I_{R1} = 0.3*0.75*20{,}000 = 4{,}500$ (please note this assumes $I_{R1}$ is proportional to $I_0$).
- Defect detection and repair costs about one labor hour each: $C_1 = 1.0$.
- Rework detection catches nine out of 10 newly created defects: $E_{V1} = 0.9$.
- Pre-delivery testing detects six defects out of 10 defects: $E_{Q2} = 0.6$.
- Test rework inserts five new defects for every 10 fixed (not the original author): $I_{R2} = 1{,}635$.
- Test rework detection catches six out of 10 newly created defects (not the original rework verifier): $E_{V2} = 0.6$.
- Test detection and repair costs 40 labor hours each: $C_2 = 40$.
- Post delivery error detection and repair costs 100 labor hours each: $C_3 = 100$.
- Ignore fixed and sunk costs: $F_1 = F_2 = F_3 = 0$.

In this case, spreadsheet analysis can be used to compute the various costs in labor hours:

- $T_{LC21} - T_{LC12} = 318{,}614 > 0$ (do not permute the inspection and testing!).
- $T_{LC1\_} - T_{LC12} = 91{,}560 > 0$ (do not drop the inspection).
- $T_{LC\_2} - T_{LC12} = 912{,}150 > 0$ (do not drop the testing).

In this case, permuting the practices raises costs, as does dropping either practice.

Just for fun, it is now possible to guess what happens when the two practices are inspection and ($Q_1$) unit test ($Q_2$). Which should one do first? Assume unit tests are 90 percent effective at finding defects, but take four hours each to find and fix the defect (additional unit tests get custom-built to diagnose and localize the defects) and verification finds 60 percent of defects created by the rework:

- $T_{LC21} - T_{LC12} = 30{,}821 > 0$ (do the inspection first).
- $T_{LC1\_} - T_{LC12} = 401{,}556 > 0$ (do not drop the inspection).
- $T_{LC\_2} - T_{LC12} = 178{,}830 > 0$ (do not drop the testing).

This was assuming unit tests were 90 percent effective. One can ask at what unit test effectiveness the two practices are neutral to interchange; in this case, approximately 55 percent. In this case, if the unit tests are less than 55 percent effective at detecting defects, they should be performed prior to the inspection.

**Note:** Please do not quote these example results! Plug in your own measurements and get real answers to your questions.

Indeed, if we use Equation 1, we can algebraically solve for interchange neutrality. On a diagram of the parameter space, the solution to this equation would divide the space into two regions, one in which the interchange is cost effective and one in which it is not:

$$(C_2-C_1)/C_3 + [(C_3-C_2)/C_3]*(I_{R1}/I_0)*(1-E_{V1})/E_{Q1}-[(C_3-C_1)/C_3]*(I_{R2}/I_0)*(1-E_{V2})/E_{Q2} = 0$$

This can be solved for $1/E_{Q2}$ directly:

$$1/E_{Q2} = \{I_0/[(1-E_{V2})*I_{R2}]\}*\{(C_2-C_1)/(C_3-C_1) + [(C_3-C_2)/(C_3-C_1)]*(I_{R1}/I_0)*(1-E_{V1})/E_{Q1}\}$$

**(Equation 4)**

provided the following inequality constraint holds:

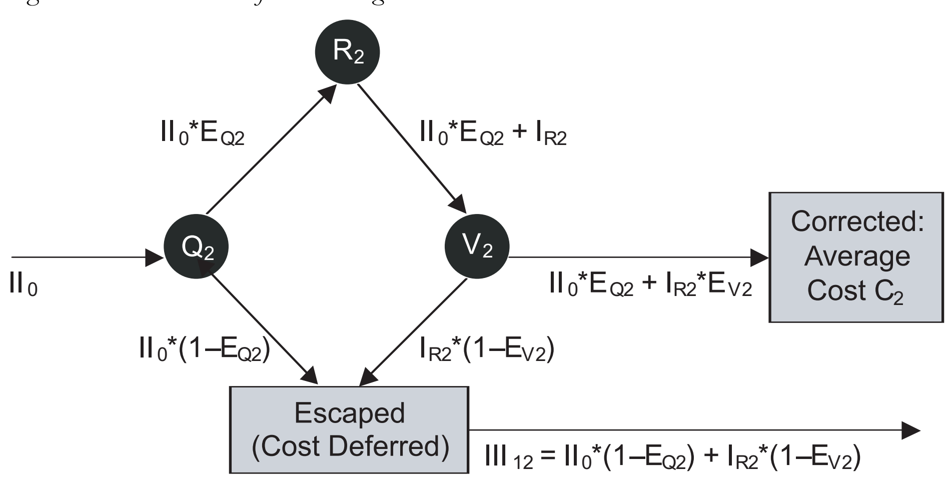
$$0 < E_{Q2} <= 1$$

## Conclusions

This analysis has demonstrated the following conclusions:

- In general, the quality practices should be ordered by increasing average cost to find and fix defects. Fixed costs do not affect this conclusion, but significant differences in either defect detection effectiveness or in the effective-

Figure 2: *Downstream Defect Handling*



ness of verifying rework induced defects can modify the conclusion.
- One should retain/add $Q_2$ provided the second practice fixes more defects during rework than the second practice creates during rework provided $C_3$ is much larger than either $F_2$ or $C_2$.

## The Measurement and Traceability Challenge

Although the above analysis does not appeal to counter-intuitive reasoning as is sometimes the case with statistical reasoning, there is a much more demanding barrier to benefiting from this analysis: getting organizations to track defects to their origin and to measure the associated costs of finding and fixing them. One problem is that quality practices are not always performed and measured the same way. Nor do they necessarily define or count defects in the same way.

To utilize the analysis presented here, each quality practice would need to measure the following items:

- C: The average cost to find and fix a defect discovered during the practice.
- $I_0$: The number of defects inserted prior to the practice.
  - o Either exclude those that are never found at all during the product life cycle – they carry no actual cost, just potential cost.
  - o Or, use a fault injection based experimental design to estimate $I_0$ with known accuracy[8].
- IR: The number of defects inserted during rework resulting from the practice.
- $E_Q$: The fraction of incoming defects ($I_0$) found by the quality practice.
- $E_V$: The fraction of defects inserted during rework ($I_R$) found during verification.

Even though there are only five items to measure, it is necessary that all quality practices use the same defect definition and that all[9] defects get traced to their point of insertion, preferably by an automated process. Effective version control and configuration management are essential here. Further, the variable costs, sunk costs, and fixed costs used in finding and fixing the defects would need to be captured.◆

## Acknowledgements

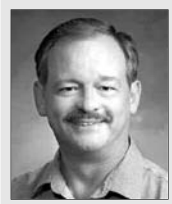
The author is indebted to Dr. Charles Farmer and Lynn Rabideau for making the time available to write the article and to his wife Carmen Lopez for tolerating a persistently scientific approach to life's common problems.

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

## Notes

1. See McCann. "How Much Code Inspection Is Enough?" CrossTalk July 2001, and "When Is It Cost Effective to Use Formal Software Inspections?" CrossTalk Mar. 2004.
2. For this analysis to be valid, it is necessary to have enough data that the concepts of *confidence interval* and *hypothesis testing* are well defined for the quantities of interest, typically mean and standard deviation. In the case of single humped distributions, required sample size is proportional to the standard deviation of the data. Statistically stable practices have less variation than statistically unstable practices, so they require less data to reach valid conclusions. The exact number of data points depends on the specific distribution being used. Analyses that is free of distribution assumptions (non-parametric analysis) typically take more, not less data.
3. The case of three adjacent practices is sufficient. The general case can be derived using the same analytic approach, although with a bit more algebraic effort. Please note that if two practices find orthogonal sets of defects, then permuting them has no effect on overall cost effectiveness.
4. Measurability of various things will depend on the process maturity of the organization. Capability Maturity Model Integration (CMMI®) Level 5 organizations will routinely address information needs related to process cost effectiveness. CMMI Level 1 organizations will be much less likely to be able to do so.
5. A fixed cost is one which does not grow in proportion to activity performed, see <http://en.wikipedia.org/wiki/Fixed_cost>. A sunk cost is one which has already been incurred and which cannot be recovered to a significant degree, see <http://en.wikipedia.org/wiki/Sunk_cost>. Although they are quite different, those differences are not relevant to this analysis.
6. See Section 804 <www.dod.mil/dodgc/olc/docs/2003NDAA.pdf#search= %22bob%20stump%20act%20of%20 2003%22>.
7. See for instance (current as of 8/2006):
   - <http://lean.mit.edu/index.php?option=com_content&task=view&id=47&Itemid=57>.
   - <www.dau.mil/conferences/2005/Wednesday/B1-1345-McElroy-CR73.pdf>.
   - <www1.eere.energy.gov/femp/pdfs/lcc_guide_05.pdf>.
   - Executive Orders 13101 and 13123.
   - OMB Circular A-94.
     - o FAR Part 7.1, especially 7.105 and Part 52.248-2 (b) and 52.248-3.
   - Defense Acquisition Guidebook, section 5.1.3.5, "Life Cycle Cost Optimization" and Chapter 3 "Affordability and Life-Cycle Resource Estimates."
8. We can put a lower bound on the probable downstream cost of undiscovered defects actually discovered later in the life cycle. Given defect injection techniques, it is actually possible to get statistically valid estimates of the number of undiscovered defects. This technique is discussed thoroughly in Mills, Harlan D. "Statistical Validation of Computer Programs" and in "Software Productivity." Dover House Publishing, 1988.
9. If a program collects enough data and has a repeatable practice of using statistical techniques in process management, then *all* can be relaxed to the idea of a *statistically significant sample*.

## About the Author



**Bob McCann** is a staff systems engineer at Lockheed Martin Aeronautics in Fort Worth, Texas. He has nearly 20 years of experience in computational physics and high performance computing, including nine years at Princeton Plasma Physics Laboratory working in the U.S. Department of Energy-controlled fusion program. McCann has served as a member of the Lockheed Martin Integrated Systems and Solution Metrics Process Steering Committee and currently works on improving systems engineering processes, methods, and metrics. He has a bachelor's degree and a master's degree in physics, a master's degree in computer science, and a master's degree in computer systems management/software development management.

**Lockheed Martin Aeronautics**
**P.O. Box 748, MZ 8605, BLDG F500**
**Fort Worth, TX 76101**
**Phone: (817) 935-4037**
**Fax: (817) 935-5272**
**E-mail: bob.mccann@lmco.com**

# Software as an Exploitable Source of Intelligence

Dr. David A. Umphress
*College of Aerospace Doctrine, Research, and Education (CADRE)*

*Security goes beyond the traditional notion of hacking into a piece of software. Software, even without being installed or used, can reveal compromising information, thus providing a security risk. This article outlines four software exploitation categories that should be considered before a software product is released.*

Our security practices tend to separate data from software. We think of data as *content*, meaning, something of operational value that can be exploited by an adversary. We tend to think of a computer program as something that performs tasks and manipulates data, not as something that has inherent informational value in itself. But we cannot escape the simple fact that software exists; it is a collection of computer instructions and supporting data. As such, it is a *thing*, something that has the potential to be broken into, taken apart, scrutinized, cannibalized for parts, or otherwise used for purposes not originally intended.

It is exactly this potential for useful information that makes software attractive for *software vulnerability attacks*. Such attacks start with software in the same form as it would be given to a legitimate user and then subjected to a number of static and dynamic tests in order to reveal compromising information. Software vulnerability attacks include the traditional notion of hacking, where a computer is broken into over a communication network, but also encompass a broad range of other tactics that view software as a source of intelligence data. Attacks need not necessarily be launched against software systems that are operational. They may be more subtle in that the attacker has physical possession of the software and is examining it for vulnerabilities under circumstances that are unobserved.

## Forms of Exploitation

Software vulnerability attacks draw from the work of software security, reverse engineering, design reclamation, and software testing to answer the question, *what does the product reveal about itself?* Unless explicitly corrected otherwise during the development phase, a piece of software has the potential to be open to four general categories of exploitation: intrusion penetration, intellectual property penetration, component penetration, and context penetration. Offering a prescription for how to identify vulnerabilities in each of these four categories is difficult – and beyond the scope of this article. However, recognizing that software is open to exploitation beyond the traditional notion of hacking is a necessary first step toward developing software processes which address holistic security.

- **Intrusion penetration** is the act of gaining illicit use of software. Someone engaging in intrusion penetration would seek to discover whether the software limits user access to functions and, if so, how securely the software deals with determining authorization. Such a vulnerability attack would analyze the software for how it authenticates users, how – or if – it encrypts data, what software features it allows users to perform, and so forth. The ultimate goal of intrusion penetration is to masquerade as a legitimate user, thus gaining access to as much functionality and data as the software offers, including, if possible, access at the level of a *super user*.
- **Intellectual property penetration** is the discovery of business rules, classified information, and protected computations encoded in software. Consider, for example, a software system that processes telemetry data from an infrared sensor in order to detect the heat signature of a missile launch. The software may well be considered unclassified when stripped of data relating to the sensor and telemetry stream, but the computer instructions themselves reveal how the data is processed, and can thus inadvertently reveal information that could be exploited. Consider also a personnel database. The structure of the database alone, absent any data, could reveal insight into what information is maintained on personnel, organizational structure, maximum size of organizations, and so forth.
- **Component penetration** addresses how the software might be used outside the context for which it was written. This form of penetration begins by discovering the individual components in the software, much as an electronics engineer would identify distinguishable components on a circuit board. Software components could then be extracted and reused in other software applications. Alternatively, software components could be extracted and replaced with substitute components that have the same interfaces but provide different functionality.

  In the first case, an adversary could obtain the use of a critical software element – such as a decryption algorithm or communication module – without having to re-create it from scratch. In the second case, the replaced component could report on the inner workings of the software, thus giving an intruder a further toehold into how the software might be further exploited programmatically. Indeed, it is not infeasible that, under certain circumstances, an intruder could intercept software being transmitted over a network and replace components with ones having nefarious purposes.
- **Context penetration** extrapolates what is known about the software under scrutiny to larger systems it may be a part of. For example, the network communication traffic that a client receives, processes, and transmits can reveal the purpose and function of its corresponding server.

It should be noted that none of the exploitation categories noted above are trivial. Most must be carried out manually though laborious examination; however, minimal attacks can uncover surprisingly revealing information, as evidenced by the following:

- **Intrusion penetration.** A software component known as a *wedge* was inserted to intercept communication between two existing software components of a large military research package. The wedge did not interfere with the functioning of the software; however, it displayed the content of data being transferred from one component to the other as the software was being executed. Through trial and error, the wedge was successfully placed at a point that revealed the license key needed to decrypt user information.
- **Intellectual property penetration.** A cursory examination of software used to track finances of a large organization revealed the toll-free access phone number, login name, and password to the organization's communication hub. While this information was not sufficient to gain access to financial information, it provided a security hole through which

an adversary could masquerade as a business unit and submit bogus data on financial transactions.

- **Component penetration.** The automatic software update mechanism was extracted from a system used to manage membership information for a national non-profit organization and was transplanted to another piece of software. In the original system, the software referenced a local XML file containing a URL and the current version numbers of the individual software modules. Using the URL, it obtained from the Web an XML file that described the most recent version of each software module, compared it to the current version, downloaded updated modules, and installed them.
- **Context penetration.** The components and configuration files of the previous system were well-enough named that their purpose was self-evident, simplifying the task of isolating the automatic update modules and adapting the configuration files for a totally different application. Although the server end of the update mechanism for the newly adapted application had to be constructed from scratch, it could be modeled after the original's client and server information interchange.

## Relevance to Today's Technology

Why are software vulnerability attacks relevant today? Harvesting information from a software artifact has heretofore been difficult and time consuming. Most products of the past have been delivered as a collection of binary executable machine instructions that mask the structure of the product and do not give easy insight into how the product might be exploited. Modern technologies that have come into heavy use in the past five years (specifically Java and .NET) have reshaped the product landscape by permitting software to be encoded using generic programming instructions. Instead of having the computer's hardware directly execute the instructions, a special program reads each hardware-generic instruction, verifies that it will not violate the computer's security policies, and carries it out as if it were part of the computer's instruction set. Since the encoded instructions are intended to be executed on any hardware platform, they must carry with them information on software module structures, data types, etc. This approach allows software to be written once and run on many different hardware platforms, thus providing on-demand delivery and installation of software to networked computers (often a necessary piece of electronic commerce and *enterprise* computing). The disadvantage is that it does so at the expense of making the software more open to analysis.

## Conclusion

Understanding software is a source of intelligence is vital to anyone involved with developing or distributing software. Of the four exploitation categories, only one, intrusion penetration, is typically examined in any depth for a given software product. Paying attention to the other categories will become more important as time goes on due to the increasingly crucial role software plays in today's economy. It is in any organization's interest – both industry and military – to minimize the amount of information exposed by software independent of it being installed or executed.◆


## About the Author

**David A. Umphress, Ph.D.,** is an associate professor of computer science and software engineering at Auburn University. He has worked over the past 25 years in various software development capacities in both industry and academia. Umphress is also an Air Force reservist, currently serving as a CADRE researcher, Maxwell AFB, Alabama. Umphress is an Institute of Electrical and Electronic Engineers Certified Software Development Professional.

**Department of Computer Science and Software Engineering**
**107 Dunstan Hall**
**Auburn University, AL 36849**
**Phone: (334) 844-6335**
**Fax: (334) 844-6329**
**E-mail: david.umphress @auburn.edu**


# CALL FOR ARTICLES


If your experience or research has produced information that could be useful to others, CROSSTALK can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for three areas of emphasis we are looking for:

**Training and Education**
*January 2008*
Submission Deadline: August 17, 2007

**Small Projects Among the Big Trees**
*February 2008*
Submission Deadline: September 14, 2007

**Distributed Software Development**
*March 2008*
Submission Deadline: October 19, 2007

**Please follow the Author Guidelines for CROSSTALK, available on the Internet at <www.stsc.hill.af.mil/crosstalk>. We accept article submissions on all software-related topics at any time, along with Letters to the Editor and BACKTALK. Also, we now provide a link to each monthly theme, giving greater detail on the types of articles we're looking for <www.stsc.hill.af.mil/crosstalk/theme.html>.**

# COTS: Commercial Off-The-Shelf or Custom Off-The-Shelf?

COTS. Everyone knows what this means, right? Commercial off-the-shelf – something you can walk into a store and buy. Well, maybe. I was recently pondering what this meant while trying to define it to a young associate. As we all know, defining things for systems and software engineers is never easy. There are always more options, parameters, and qualifiers. Let's look at a few examples of *commercial off-the-shelf* purchasing.

Since I drive an ancient rusting truck, I've been frequenting car dealerships to find new transportation. It is easy to walk by all the shiny new cars on the lot and think *yes*, I can buy one of these, right off the lot. Much to my dismay, after appropriately discounting the asking price and reconciling with my monthly budget, I needed something not quite off the shelf. After some negotiations on what features I wanted, it became obvious that it was best that I get more features than I needed to get just what I wanted. Base car + package A + options B, C, and D = my perfect car, right off the shelf, for a price, and oh, can I wait six weeks for it? I left the dealership with my head spinning from all of the choices, still without a COTS car, but with my bank account intact.

After such a grueling ordeal at the car lot, I decided maybe it was time for some lunch from my local sandwich shop. I wanted the sandwich in picture number one, nothing special, just a basic sandwich. What type of bread? What meat? What cheese? What crunchy stuff did I want? What dressings? So much for off-the-shelf. I could have chosen a number one and then completely changed my sandwich by making different choices along the way. I began to think maybe the sandwich shop wasn't the best place to look for commercial standardization. As I left the sandwich shop, I realized that *commercial off-the-shelf* might not be as standard as the connotation of the phrase implies. It is more like *custom off-the-shelf*.

Of course, cars and sandwiches don't compare to weapon systems, do they? Yet they are closer than the military industrial base would like to admit. In today's world, the commercial world is going towards more and more customization rather than a standard product off the shelf. When you bought your last personal computer, did you go buy one off the shelf of the local store or did you go to a Web site and click through pages and pages of options? What color of MP3 player do you want? Everyday items from cars to toasters are now customized. It is as if the commercial industry realized that the military got it right – customization is good.

What the military really wants are basic capabilities with lots of options to customize their materiel at an affordable price. The affordability is where commercial industry surpasses the military industry. The additional costs to customize my car and my sandwich were minimal. The custom car I almost bought would be repaired in the same shop as other cars from the same manufacturer, and at a standard labor rate. Of course, the military likes to think they have unique requirements over and above those required by commercial industry. While there is some truth to this, most of the military equipment is now coming up to commercial standards, rather than commercial components coming up to military standards. While standards are good, rarely does an entire weapon system fit in any one standard. Each component or subsystem may comply with industry standards but the components are then custom-integrated into a usable weapon system.

So, why is the military pursuing COTS? I think it is because they really want to customize their purchase, hoping that they can get it at affordable COTS pricing. Also, they think that COTS will speed up the process. If there's anything that this issue of CrossTalk teaches us, it's that COTS sometimes saves neither time nor money.

See, customizing COTS takes extra time and extra money. You want to buy an off-the-rack suit? If you're exactly a 44R coat and can wear 38 pants with a 29" inseam, no problem. However, just the slightest change in either suit or pants size will not only cost more, it will *dramatically* increase the *out-the-door* time. Instead of wearing your suit home, you have to wait a week or so for customization – same with COTS. You buy it to save time and money. However, unless it fits you *exactly*, you will spend additional time and money getting it customized to fit your needs. In fact, you might also have to customize the other software that the COTS interact with to make it *fit*; this costs even more money and even more time. In the Department of Defense, sometimes the time is more important than the money – and customizing COTS is typically a very slow process.

So, how do you want your COTS customized?

—Wiley F. Livingston, Jr. P.E.
*USAF, 580 SMXG/Flight C Chief*
wiley.livingston@robins.af.mil

P.S. And if you lose or gain weight, and your suit size (and COTS needs) change … well, let's not even go there!

## Can You BackTalk?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CrossTalk, we also accept articles for the BackTalk column. BackTalk articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BackTalk article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BackTalk article, visit our Web site at <www.stsc.hill.af.mil>.


CrossTalk is co-sponsored by the following organizations:

NAVAIR

Homeland Security

**CrossTalk / 517 SMXS/MXDEA**
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737